Timeless Planning and the Component Placement Problem

Tatiana Kichkaylo
New York University
New York, NY 10012, USA
kichkay@cs.nyu.edu

Abstract

Planning is traditionally associated with the search for sequences of actions spread over time. In this paper we present a component placement problem (CPP), which is concerned with sequences of actions spread in the space of wide-area networks. We argue that, despite the lack of a time component, the CPP is a planning problem. We discuss complexity of the CPP and similarities between the CPP and traditional AI planning.

1 Introduction

The component-based approach to application development is widely used in areas involving large-scale distributed applications. Adaptive component-based systems, web services, grid computing all are concerned with applications consisting of components running on different computers and communicating over the network. A *component deployment* on network N is a DAG of such communicating components and an assignment of the components to nodes of N and of communication channels between components to links of N. The *component placement problem* (CPP) is the problem of finding at run-time a deployment that enables a given application to run well, satisfying both qualitative and quantitative constraints, given the current state of the network.

The complexity of the CPP and the choice of an algorithm to solve it depend on the model used to specify application components and their resource requirements.

- The components may have single or multiple inputs and outputs. This determines whether the application configuration is a chain or a DAG.
- The components may interact with each other by passing continuous streams of data or through explicitly represented intermediate results (typically stored in files). In the former case the application configuration is a snapshot, in the latter the application is distributed in both network space and physical time.
- The underlying network may support explicit reservation of resources or provide best-effort service. As a consequence, the amount of resources available on network nodes and links may be a constant or vary as a function of time.
- The resource requirements and running time of components may be specified as constants; as linear functions of available resources and data being processed; or as arbitrary functions. Some models of the CPP, e.g. one-to-one components with linear functions for resource consumption, allow for fast algorithms. More general models do not promise easy solutions.

In our previous work (Kichkaylo, Ivan, & Karamcheti 2003) we have presented a model for the CPP that involves stream processing components with an arbitrary number of inputs and outputs and arbitrary monotonic resource functions; and we have presented an algorithm, called Sekitei, to solve the CPP in this model.

In this paper we discuss a general version of the CPP in the context of AI planning. First, we review the basic model used in Sekitei. Second, we show correspondence between the basic model and AI planning. Although the basic model of the CPP is concerned with stream-based applications and therefore with snapshot (*timeless*) configurations of application, we show that it is very similar to traditional AI planning problems involving time. In Section 4 we discuss more general models of the CPP. Then we discuss techniques for solving the CPP and present related work.

2 The Basic Model of the CPP

A component placement problem is concerned with finding a valid *component deployment*, i.e. a set of components, linkages between them, and a mapping of the resulting DAG onto links and nodes of a wide-area network, such that client requirements are satisfied.

The model of the CPP used in Sekitei is that components transform data streams, referred to as *interfaces*, produced by a server component(s) and consumed by the client component. First, a DAG of components linked by interfaces needs to be constructed, whose endpoints are the client and a subset of existing servers. This DAG is then mapped onto the network graph in such a way that all resource constraints are satisfied. The algorithms mentioned in this paper perform both these actions (the DAG construction and the mapping)

simultaneously.

The specification of the CPP consists of

Specification of the network configuration and resources. The network consists of nodes and links between them. Various real-valued properties, such as the amount of physical memory, link bandwidth, network latency, might be associated with the links and nodes.

Specification of interfaces. For each interface type, the link crossing behavior is specified as a set of assignments describing change in link properties (e.g. bandwidth consumption) and properties of the interface on the destination side of the link as functions of original link properties and properties of the interface at the source.

Specification of components. A component is specified by zero or more required interfaces, zero or more implemented interfaces, component placement requirements (a conjunction of inequalities involving functions over node resources and properties of required interfaces), and component placement effects.

The goal of the CPP is to place a component of a given type on a given network node.

For example, consider the following webcast application (Figure 1).

- The network consists of six nodes belonging to two LANs with high-bandwidth links between the nodes within each LAN. The link between the LANs (between nodes 2 and 4) has low bandwidth. Each node has some number of available CPU cycles (per unit time).
- There are four types of interfaces in this application: media stream (M), image stream (I), text stream (T), and zipped text stream (Z). The only property we care about is bandwidth of the interface, which is related to the rate of requests. As a result of crossing a network link, bandwidth of the interface at the destination is computed as minimum of the link bandwidth and bandwidth of the interface at the source, and the link bandwidth is decreased by the same amount.
- There are five types of components in this application. The client component (Cl) consumes (requires) a high-bandwidth media stream. The splitter component (Sp) accepts a media stream and splits it into text and image streams. The sum of bandwidths of the two resulting streams is equal to the bandwidth of the original media stream. The merger component (Mr) performs the reverse transformation. The zip (Zp) and unzip (Un) components compress and decompress a text stream. The bandwidth of the resulting zipped stream is significantly smaller than that of the text stream. All components consume CPU cycles when deployed on network nodes.
- The goal in this problem is to place a client component on node 1 given that a high-bandwidth media stream is available on node 6 (i.e. there is a server (Sr) component running there).

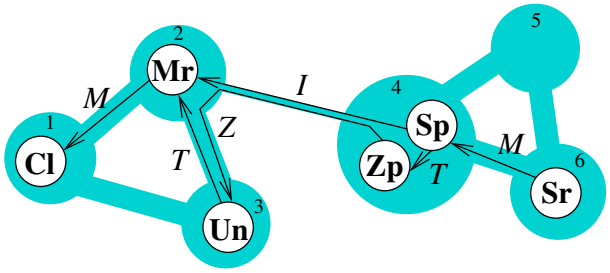

Figure 1: The webcast application.

If there is a network path between nodes 1 and 6 with sufficient bandwidth, the client can be directly connected to the server. However, in more resource-restricted situations a solution to the CPP would involve injection of additional components into the network as shown on Figure 1. In general, the choice of components involved in a particular deployment depends both on requirements of components and on resources available in the network.

3 CPP and Traditional Planning

The CPP can be compiled into a planning problem by introducing operators for component placement and link crossing (Kichkaylo, Ivan, & Karamcheti 2003). This compilation produces a planning problem without negative logical preconditions and effects, but with very general resource functions.

In this section we show that, despite the lack of negative boolean values, the CPP is at least as hard as propositional STRIPS planning. On the other hand, the CPP has more structure than the general planning with metric resources, which is known to be undecidable (Helmert 2002). This gives us a hope of constructing efficient algorithms for the CPP.

3.1 Compilation of CPP into a planning problem

The CPP can be mapped onto a planning problem with metric resources as follows.

Objects of the planning problem correspond to network nodes, component types, and interface types. There are three predicates: `av(?interface ?node)` stating that an instance of the interface type is available on the given network node,[1] `placed(?component ?node)` stating that an instance of the component is deployed on the node, and `link(?node ?node)` describing existence of a network link between two nodes. In addition, for each resource type used in the CPP, the

[1] We assume that there is at most one independent stream of each interface type on a network node. Dependent streams, e.g. sensor inputs produced by different sources and consumed by a single aggregating component, can be modeled by adding a `counter` property to the interface.

```
(:action placeZp
 :parameters (?n - node)
 :precondition (and (av T ?n)
                    (>= (cpu ?n) (/ (ibw T ?n) 10)))
 :effect (and (placed Zp ?n) (av Z ?n)
              (assign (ibw Z ?n) (/ (ibw T ?n) 2))
              (decrease (cpu ?n) (/ (ibw T ?n) 10)))
)
```

Figure 2: An operator for placing a zip component.

planning problem has a real-valued fluent. For example, if only CPU and bandwidth resources are considered, the planning problem has functions `cpu(?node)` for available CPU on a node, `lbw(?node ?node)` for bandwidth of a network link between two nodes, and `ibw(?interface ?node)` for bandwidth of an interface on a node.

The planning problem has two types of operators. For each component type, `place_`*component*`(?node)` describes placement of a component instance onto a node. The logical preconditions of this operator include availability predicates for all interfaces required by the component. The logical effects enumerate availability predicates for all implemented interfaces. Resource preconditions of the `place` operator describe placement preconditions, and resource effects are obtained from the resource consumption formulas.

For example, the operator for placing a zip component is shown on Figure 2. `placeZp` can be executed on a node that has an instance of the text stream and sufficient CPU resources. As a result, an instance of the zipped stream becomes available on that node with bandwidth computed as a function of the bandwidth of the incoming text stream, and some CPU cycles are consumed.

The second operator, `cross_`*interface*`(?node ?node)`, describes sending an instance of the interface over a network link.[2] Its preconditions include availability of the interface at the source, and the existence of the link. The logical effect of the operator is that the interface becomes available at the destination. The resource effects describe properties of the interface at the destination and change in the properties of the link. Figure 3 shows an example of such operator.

Although in the above example we use simple analytical expressions to describe resource functions, in general, no properties of resource functions are assumed. For example, resource consumption of a component may be described by a (non-polynomial) approximation based on profiling results.

The initial state of the planning problem is obtained from the state of the network and contains `av` predicates and initial values for metric resources. The goal is usually represented as a single `placed(?component ?node)` predicate. However, the planner discussed in Section 5 can handle any conjunction of `placed(?component ?node)` and `av(?interface ?node)` predicates.

[2]We need separate `cross` operators for each interface type, because different interfaces might have different sets of properties and different link crossing behavior.

```
(:action crossM
 :parameters (?from ?to - node)
 :precondition (and (av M ?from) (link ?from ?to))
 :effect (and (av M ?to)
    (assign (ibw M ?to)
            (min (ibw M ?from) (lbw ?from ?to)))
    (decrease (lbw ?from ?to)
            (min (ibw M ?from) (lbw ?from ?to))))
)
```

Figure 3: An operator for sending M stream over a network link.

This compilation of the CPP produces a planning problem without negative logical preconditions and effects. None of the operators requires absence of interfaces, or removes components or interfaces. If an interface is completely consumed, its available bandwidth simply becomes zero. We do not pose any restrictions on the form of functions used in representation of operators, except that they have non-negative values.

3.2 The CPP and STRIPS planning

Intuitively, a state of the world in the traditional planning corresponds to a state of an interface in the CPP. The following reduction makes this statement more formal.

A propositional STRIPS planning domain is described by a set of boolean variables $P = \{p_i\}$, a set of actions $A = \{a_j\}$, where each action is defined by a list of preconditions (a conjunction of a subset of the variables or their negations), an add list (a set of variables that become true as the result of the action execution), and a delete list (the set of variables that become false). A STRIPS planning problem is defined by a planning domain, an initial state (a complete truth assignment to all variables), and a goal state (a partial truth assignment). The goal of the planner is to find a (totally ordered) sequence of actions that when executed in the initial state brings the system to a goal state.

A propositional STRIPS problem is identical to the following CPP problem. The network contains only one node, and no resources are associated with this node. There is only one interface type with $|P|$ properties. Initially, this interface is available on the node. For each variable p_i, the value of the corresponding property is 1 iff p_i is true in the initial state. For each action a_j, the CPP contains a component that consumes and produces the interface. The placement precondition of this component is a conjunction of equalities corresponding to the action's preconditions. The effects of the component placement are assignments of 0 or 1 to some properties of the interface (in accordance to the add

and delete lists) and identity assignments to the rest of the properties. The identity assignments correspond to the frame assumption in STRIPS that properties not in the add or delete list remain unchanged. An additional component C_{goal} is constructed with placement preconditions generated from the goal state description. The goal of the CPP is to place C_{goal} on the node.

For example, consider the following problem. The world state is described by three boolean variables p, q, and g, all of which are initially false. There are three actions:

action	precondition	add	delete
opA	none	p	none
opB	none	q	p
opC	$p \wedge q$	g	none

The goal is to achieve g.

This problem maps to the following CPP. Initially, there is interface Int available on the node 0 with three properties set to 0: $Int.p(0) = 0, Int.q(0) = 0, Int.g(0) = 0$. There are four component types, each of which consumes and produces interface of type Int. The resource requirements and effects of the components are given in the following table. Values on the left side of the effect assignments refer to the produced data stream, and those on the right side of the assignments to the consumed stream.

component	requirements	effects
compA	none	$Int.p(n) = 1$; $Int.q(n) = Int.q(n)$; $Int.g(n) = Int.g(n)$
compB	none	$Int.p(n) = 0$; $Int.q(n) = 1$; $Int.g(n) = Int.g(n)$
compC	$Int.p(n) = 1$, $Int.q(n) = 1$	$Int.p(n) = Int.p(n)$; $Int.q(n) = Int.q(n)$; $Int.g(n) = 1$
C_{Goal}	$Int.g(n) = 1$	$Int.p(n) = Int.p(n)$; $Int.q(n) = Int.q(n)$; $Int.g(n) = Int.g(n)$

The goal is to place C_{Goal} on node 0.

The solution for this CPP is to place a chain of components *compB*-*compA*-*compC*-C_{Goal} on the node. This CPP solution corresponds to the sequence of actions `opB`, `opA`, `opC`, which is a solution for the original STRIPS problem.

The above polynomial reduction proves that the CPP is at least as hard as propositional STRIPS planning, namely, PSPACE-hard (Bylander 1991). However, we do not believe that a reduction in the opposite direction is possible. The CPP puts no restrictions on the form of resource functions, which prevents it from being compiled into a boolean-only STRIPS problem.

3.3 The CPP and resource planning

The reduction of STRIPS planning to CPP does not use resource functions beyond constant assignments and comparisons. In general, metric planning in the presence of sufficiently complex expressions involving resources is undecidable (Helmert 2002). However, when comparing the CPP with resource planning, two points are worth making.

First, the CPP has only two types of actions – component placement and link crossing – and does not have negative preconditions and effects. However, in the presence of resource functions, this restriction does not affect the complexity of the problem, since propositional variables can be replaced with resource variables taking only values 0 and 1 as illustrated by the STRIPS to CPP reduction.

Second, the CPP does not specify an upper bound on the number of components that can be placed on a node or the number of streams that can be sent over the same network link. This means that, although the network topology and the set of component types are finite, the plan length is unbounded. However, in reality, every action usually consumes at least one non-replenishable resource. For example, sending an interface (data stream) over a network link consumes link bandwidth, and a component placed on a network node consumes CPU cycles. Because of this consumption of non-replenishable resources, the total number of actions that can constitute a plan is limited.[3] This means that, in practice, it may be possible to determine existence or non-existence of a solution for the CPP in finite time.

3.4 Network paths and time lines

The planning problem resulting from compiling a CPP into a planning problem can be viewed as a "normal" planning problem with resources. This problem has discrete time steps corresponding to evolution of a data packet through a data path (e.g. processed by a component, *then* transmitted to another node over a network link). In the model of the CPP described so far, the exact delay of a packet from its origin is viewed as one of the properties of the interface (similar to bandwidth), and the problem specification can define any functions for manipulating delays. Below we will describe an extension of the model, where time is reintroduced as a first class object and treated in a more traditional way. In this section we would like to point out that the "space" aspect of the CPP is closely related to the "time" aspect of the traditional planning.

Transformation of a data stream by components and links along a data path is similar to transformation of a world state by actions along the time line. Different data paths can affect each other via shared network resources just like actions in classic planning interact via shared variables. For example, in the CPP it is possible to put two components processing different data streams onto the same network node. This situation corresponds to parallel execution of actions in tradi-

[3] Assuming that there is some minimum bound on the resource consumption, so that the system does not experience Zeno's paradox.

tional planning. In both cases, if the sum of resource requirements of components/actions exceeds the available amount of resources, a conflict occurs. Techniques for resolving conflicts in classic planning, such as promotion and demotion, correspond to changes in a data path in the CPP, e.g. by sending a data stream to another node. Even though the CPP does not have an explicit time component, it can be considered *"planning in space"* similar to the traditional *"planning in time"*.

There are, however, important differences. The time considered by traditional planning is homogeneous. A logical state enables the same set of actions regardless of at what point along the time line this state occurs. In the CPP, different nodes and links have different properties. Therefore, different sets of components may be deployed on different nodes even if exactly the same data streams are available on these nodes.

Moreover, the time in planning is linear. For each step there is exactly one step immediately preceding it and one step immediately following it. On the other hand, the network in the CPP is a graph. There are several paths leading to and from each node.

3.5 The CPP and scheduling

In the CPP, a data stream may be forced to "jump" to a new node because of limited resources at the source node. Nodes and links in the CPP are resources in the scheduling terminology.

In scheduling, the problem is to allocate a given set of resources to a given set of tasks. In the CPP the set of components constituting an application configuration is not fixed. A planner may decide to use a caching component to deal with a low-bandwidth path, a pair of zip/unzip components, or choose a longer network path. This need for choosing an action and the fact that the plan length cannot be computed in advance prevent us from directly using constraint satisfaction techniques for solving the CPP.

In scheduling, it is often trivial to find a feasible solution by stretching the schedule over time. In the CPP as it is presented in Section 2, many resources, such as network bandwidth, are consumable. Stretching the component DAG over the network simply replaces one resource conflict with another, and often does not lead to a solution. Instead, one needs to change the DAG itself, i.e. the set of components participating in the deployment. Therefore, the problem of finding a feasible solution for a CPP requires solving the plan existence problem in the presence of resource constraints.

4 Extensions to the CPP

The basic version of the CPP presented in Section 2 is concerned with finding a feasible configuration of a stream-processing component-based application. In this formulation, metric network resources such as bandwidth and CPU are assigned in a greedy fashion. The stream model of applications also means that the time aspect is completely ignored. Although it is possible to reason about the delay property of a data stream with respect to the server, the model does not accommodate use of the same network resources by different components at different time points.

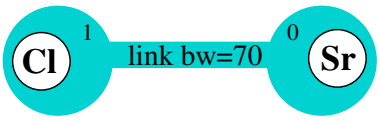

Figure 4: Webcast scenario in which the greedy approach does not find a solution.

In this section we present two important extensions to the basic model of the CPP and show how they relate to traditional planning problems.

4.1 Optimization of resource consumption

The definition of the CPP given in the beginning of this paper describes resource consumption as driven by the server side. In practice, the resource consumption also depends on the client. For example, in our webcast application, if the client is not able to consume the data at the maximum rate supported by the server, the data packets will be delayed in the transmission queues of transit network nodes, thus limiting the rate at which the server issues the data.

Our original planner (Kichkaylo, Ivan, & Karamcheti 2003) relied on this self-stabilization property and used a greedy approach to resource allocation. However, as our experience shows, it is sometimes crucial to optimize the resource consumption explicitly. For instance, in the following example the greedy approach cannot find a solution at all.

Suppose, we want to deliver at least 90 units of bandwidth of the M stream (the requirement of the client component) over the link with bandwidth 70 (Figure 4). The source node 0 has 200 units of M available, but only 30 units of CPU. Suppose, transformation of 200 units of M by the splitter requires 40 units of CPU. Sending the M stream directly to the client does not satisfy client's bandwidth requirements, and the amount of CPU available on node 0 is less than required for processing all available bandwidth of the M stream by the splitter, therefore, the greedy approach will not find a solution. If, on the other hand, we allow the splitter to transform only 90 units of bandwidth of the available M stream, then the total CPU requirements of the splitter and zip components may be less than 30, and the following solution can be found:[4]

```
place Splitter on node 0,
place Zip on node 0,
cross with Z stream from node 0 to 1,
cross with I stream from node 0 to 1,
place Unzip on node 1,
```

[4]We assume that the client node has sufficient amount of resources for unzip and merger components.

```
place Merger on node 1.
```

The same issue arises in more traditional settings. For example, consider the problem of students driving from one city to another. To achieve their goal, the students need both to get to the destination, which requires buying gas for the car, and have lunch. A situation is easily imaginable when, given a restricted budget, it is not possible to buy both a lunch and a full tank of gas, but partial refueling would be enough to achieve the goal.

The above problems essentially require real-valued action parameters, which are not allowed by the PDDL standard, but are already supported by some planners (Shin & Davis 2004). Such real-valued variables also allow for expressing preferences by incorporating these resource variables in action cost functions.

4.2 Planning in space *and* time

A natural extension of the CPP is construction of deployments for grid and web service applications, where reasoning about delays and sharing of resources in time is essential.

In this more general version of the CPP, the network topology is static, but the availability of metric resources on links and nodes is a function of time. For example, in a scientific application, one of the components may require a particular type of a supercomputer to run on. Such equipment is usually in high demand, and a time slot needs to be reserved for the application.

Availability of time slots, relative speeds of computers at network nodes, delays involved in transferring data files over network links can all affect the solution. Given relatively high cost of network bandwidth and transfer times, a good solution might require recomputing of some data files even if a copy of that data is already available in the network.

The version of the CPP with explicit reservations can be compared to planning in the presence of external events. As before, although this problem can be compiled into traditional planning with metric resources, knowledge of the structure of the problem might help to create more efficient algorithms.

5 Solving the CPP

Any version of the CPP discussed above can in principle be compiled into a traditional planning problem with metric resources. A number of modern planners support metric resources (Do & Kambhampati 2003; Hoffmann 2002), but the inherent complexity of the problem prevents them from scaling well.

In the case of the CPP, scalability is an extremely important issue. Even though the total number of components participating in any realistic application deployment is usually small, the size of the problem specification is often huge. In principle, the planner can use any network link in the Internet while designing a path to send a data stream from a server to a client. In grid applications, there are often many replicas of the same data file located on different nodes in the network, and it is hard to say a priori which copy should be used in a particular scenario.

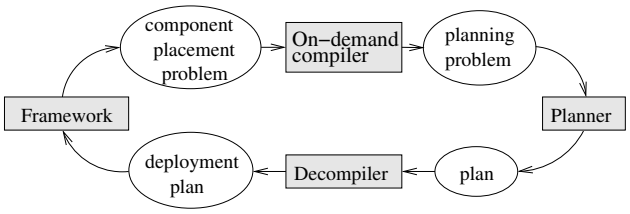

Figure 5: Integration of the Sekitei planner with the framework.

5.1 Solving the basic CPP

Good scalability can be achieved by exploiting the structure of the CPP. In (Kichkaylo, Ivan, & Karamcheti 2003) we described the Sekitei planner for the basic CPP. Sekitei adds only one restriction to the definition of the problem given in Section 2: All resource functions used in resource preconditions and effects should be monotonic with respect to each parameter. The planner combines several techniques to achieve good scalability on the CPP in the presence of arbitrary monotonic resource functions.

Sekitei starts by performing regression based on logical part of the problem specification. The regression phase of Sekitei considers propagation of availability of interfaces through the network nodes similar to traditional planners propagating fluents through time, thus taking advantage of the time/space flip discussed in Section 3.4.

The core of our planner uses ground instances of operators, propositions, and resource functions. To avoid memory explosion during generation of ground instances, the planner is cushioned from the network by an on-demand compiler (Figure 5). The compiler creates ground instances of the `place` and `cross` operators in response to requests from Sekitei of the type "how can I achieve availability of the given interface on the given node". This is easy to do, because interface *Int* on a node n can be produced either by placing a component implementing interface *Int* on node n or by delivering interface *Int* over a network link that has node n as one of its endpoints. To obtain necessary information about network topology and available components, the compiler uses external services.

Results of the regression phase, namely the set of ground instances of operators and propositions relevant for achieving the goal, is used by the progression phase of Sekitei. During this stage, aggregated resource intervals are pushed along the identified portions of the network similar to GraphPlan-based algorithms (Koehler 1998). The progression stage relies on the monotonicity of the resource functions and can be thought of as application of resource envelopes (Laborie 2003) to planning in space.

Initially, the regression phase proceeds until the (logical part of the) initial state is reached. Since the progression phase takes into account resources, it is possible that it fails for the minimum regression graph. In this case Sekitei expands the regression graph and tries again.

The plan extraction phase of Sekitei is again based on GraphPlan, but uses additional memoization techniques to deal with the resource functions (Kichkaylo, Ivan, & Karamcheti 2004).

The layered approach allows Sekitei to achieve good performance on the basic CPP. The use of progression permits Sekitei to reason about non-reversible resource functions, while the combination of regression and on-demand compilation prevents explosion on large problems. More details on Sekitei and its performance are provided in (Kichkaylo, Ivan, & Karamcheti 2004).

5.2 Supporting extensions

Although the Sekitei algorithm described above cannot be directly used for the extended versions of the problem discussed in Section 4, many ideas and techniques are reusable.

We have recently developed a new version of our algorithm capable of dealing with cost functions (Kichkaylo & Karamcheti 2004). The original Sekitei algorithm supports non-reversible functions. In this model the outputs of a component are computed as functions of its inputs, but it is impossible to compute inputs given outputs. For example, it is not possible to compute bandwidths of text and image streams consumed by a merger component given only desired bandwidth of the produced media stream. If we assume that all functions are reversible, our new algorithm can guarantee optimality of a solution with respect to a cost function. In the presence of non-reversible functions this behavior can be approximated using discrete levels of resources supplied as a parameter to the planner.

Both algorithms mentioned above assume that resource availability does not change over time. This model is adequate for describing stream-based and best-effort file-based grid applications (e.g., the Condor model (Thain, Tannenbaum, & Livny 2002)). The need for explicit reservations of resources adds a new dimension to the problem. Development of an efficient algorithm for the CPP with explicit reservations is a challenging research problem.

6 Related Work

Several modern planners solve the general planning problem with metric resources (Hoffmann 2002; Shin & Davis 2004; Do & Kambhampati 2003; Wolfman & Weld 2001). However, as expected given the complexity of the problem, they are not very scalable. In addition, traditional planners often limit themselves to linear expressions in resource functions, which is not sufficient for many realistic instances of the CPP.

Existing planners for the basic CPP developed in the systems community support only special cases of the problem, such as chains of components (Fu & Karamcheti 2003) or pre-existing deployments (Gribble *et al.* 2001).

Pegasus (Blythe *et al.* 2003) is a system for deployment planning for grid services. To the best of our understanding, it does not support explicit reservations, and therefore falls into the basic CPP category.

We believe (and our belief is supported by the performance of Sekitei) that a planner for the CPP needs to take into account the structure of the problem. In particular, the search techniques need to focus on resources. The various limited consistency techniques developed by the constraint reasoning community might be of great use here.

For some planning problems involving metric resources it is beneficial to separate planning (action selection) and scheduling (resource assignment) (Srivastava 2000). The CPP is an example of the problem where such separation would lead to degraded performance, because the action selection in this case is *driven* by resource constraints. Algorithms for the CPP that focus on resource constraints might be useful for solving other problems having similar structure, for example, in the logistics and space domains.

7 Summary

In this paper we have presented the component placement problem. The CPP is concerned with finding a configuration of a distributed application in a wide area network. The model assumed by the basic CPP is that of data streams processed by software components, and talks about a snapshot of the network state without any mention of time. However, we believe that, despite this timelessness, the CPP is a planning problem. We have discussed correspondence between traditional AI planning and the CPP, and provided preliminary complexity results.

There are several important extensions of the basic CPP. We have discussed two of them, including explicit reservations and cost functions dependent on resource consumption. Although the CPP can be compiled into a traditional planning problem with metric resources, more efficient algorithms can be constructed if the knowledge of the structure of the CPP is used.

The CPP shares some features with other real-world planning problems involving resources, for example, in the logistics and aerospace domains. The resource-focused search techniques developed for the CPP are applicable to these problems as well.

8 Acknowledgments

The author is grateful to Ernest Davis for helpful discussions. This research was sponsored in part by NSF grants IIS-0097537, CAREER:CCR-9876128, and CCR-0312956; by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929.

References

Blythe, J.; Deelman, E.; Gil, Y.; Kesselman, C.; Agarwal, A.; Mehta, G.; and Vahi, K. 2003. The role of planning in grid computing. In *ICAPS*.

Bylander, T. 1991. Complexity results for planning. In *IJCAI*.

Do, M., and Kambhampati, S. 2003. Sapa: A scalable multi-objective metric temporal planner. *JAIR*.

Fu, X., and Karamcheti, V. 2003. Planning for network-aware paths. In *Proc. of DAIS*.

Gribble, S.; Welsh, M.; von Behren, R.; Brewer, E.; Culler, D.; Borisov, N.; Czerwinski, S.; Gummadi, R.; Hill, J.; Joseph, A.; Katz, R.; Mao, Z. M.; Ross, S.; and Zhao, B. 2001. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks* 35(4):473–497.

Helmert, M. 2002. Decidability and undecidability results for planning with numerical state variables. In *AIPS*.

Hoffmann, J. 2002. Extending FF to numerical state variables. In *ECAI*.

Kichkaylo, T., and Karamcheti, V. 2004. Optimal resource-aware deployment planning for component-based distributed applications. To appear in *HPDC*.

Kichkaylo, T.; Ivan, A.; and Karamcheti, V. 2003. Constrained component deployment in wide-area networks using AI planning techniques. In *IPDPS'03*.

Kichkaylo, T.; Ivan, A.; and Karamcheti, V. 2004. Sekitei: An AI planner for constrained component deployment in wide-area networks. Technical Report TR2004-851, NYU.

Koehler, J. 1998. Planning under resource constraints. In *ECAI-98*, 489–493.

Laborie, P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence* 143(2):151–188.

Shin, J., and Davis, E. 2004. Continuous time in a SAT-based planner. In *AAAI-04*.

Srivastava, B. 2000. Realplan: Decoupling causal and resource reasoning in planning. In *AAAI*.

Thain, D.; Tannenbaum, T.; and Livny, M. 2002. Condor and the grid. In *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc.

Wolfman, S., and Weld, D. 2001. Combining linear programming and satisfiability solving for resource planning. *The Knowledge Engineering Review*.