Building Self-configuring Services Using Service-specific Knowledge

An-Cheng Huang[†] and Peter Steenkiste[†‡]
Computer Science Department[†] Electrical and Computer Engineering Department[‡]
Carnegie Mellon University

Abstract

A self-configuring service can automatically leverage distributed service components and resources to compose an optimal configuration according to both the requirements of a particular user and the system characteristics. One major challenge for building such services is how to bring in service-specific knowledge, e.g., what components are needed and optimization criteria to use, while still allowing reuse of common service composition functionalities. In this paper, we present an architecture in which service developers express their service-specific knowledge in the form of a service recipe *that is used by a generic* synthesizer *to perform service composition automatically. We apply our approach to three different services to illustrate the flexibility and simplicity of the recipe representation. We use simulations based on Internet measurements to evaluate how an appropriate optimization algorithm can be selected according to a developer's service-specific trade-off between optimality and cost of optimization.*

1. Introduction

Infrastructures such as Globus [4] and the Open Grid Services Architecture [5] enable dynamic discovery, monitoring, and deployment of distributed resources (e.g., computation and storage servers) and service components (e.g., video transcoder and multicast proxy). This allows developers to leverage service components and resources to build user-level services such as scientific visualization [16], distributed simulation [15], and video conferencing [11]. One promising direction for developing such services is *self-configuration*. Instead of statically integrating components when the service is developed, self-configuring services dynamically combine available components and resources into optimized service configurations at the invocation time, i.e., when a user request is received. This allows the service composition process to take the particular user requirements and the system characteristics into consideration.

This research was sponsored in part by the Defense Advanced Research Project Agency and monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518. Additional support was provided by Intel.

Recent research efforts to support dynamic service composition can be divided into two categories. *Service-specific* architectures are designed for particular classes of services/applications [3, 15, 9, 19, 16]. Examples include resource selection for resource-intensive applications and resource allocation for services consisting of a set of multi-fidelity applications. Others propose *generic* architectures that can compose different services using "type-based composition" [8, 23, 20, 6, 13]. Components have well-defined input/output (requires/provides) interfaces, so a service composition module can automatically generate a service configuration providing the requested interface(s).

One key difference between these two approaches is the use of *service-specific knowledge*. Such knowledge can greatly reduce the space of feasible configurations and provide service-specific optimization criteria for selecting components/resources. In the service-specific approach, the service-specific knowledge is hard-wired into a service composition module by the developer, so it is very difficult to leverage this effort in other services. On the other hand, although the generic architectures can be reused by a broad range of services, they cannot exploit service-specific knowledge, often resulting in higher overhead and sub-optimal configurations, e.g., using generic optimization criteria such as minimizing the number of components.

In this paper we present an architecture for dynamic service composition that strikes a balance between these two approaches: our approach is general, yet developers can introduce service-specific knowledge into the composition process. The key elements of our architecture are the *service recipe*, which expresses a developer's service-specific knowledge, and the *synthesizer*, which performs the service composition. When a user request is received, the synthesizer uses the knowledge in the recipe to automatically perform service composition according to the specified user requirements and the system characteristics. Unlike the service-specific architectures, our architecture provides generic service composition functionalities that can be reused by developers of different services. In contrast to the generic architectures, developers can express their composition heuristics and optimization criteria in recipes

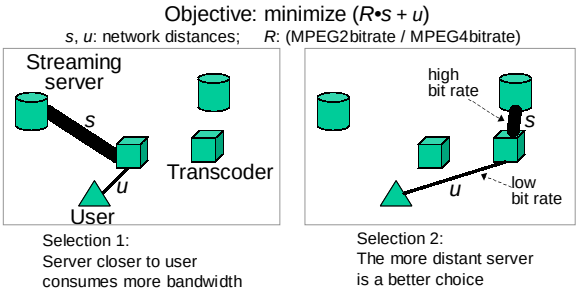


Figure 1. A video streaming service

to customize how service composition is performed.

Our focus in this paper is on how service-specific knowledge can be represented in a recipe that can be used by a general synthesizer. Sections 2 and 3 describe the dynamic service composition problem and application examples and discuss previous efforts in this area. Section 4 presents our architecture and describes how service-specific knowledge is used to compose service configurations. Section 5 describes the physical mapping problem and discusses how the synthesizer can select the best algorithm according to service-specific trade-off. We conclude in Section 6.

2. Problem statement

Let us use a video streaming example to illustrate our service model. A *service developer* who has expertise in video streaming builds a self-configuring video streaming *service* in which the service-specific knowledge is embedded. A *user* who wants a low-bitrate MPEG-4 video stream submits a *request* to the self-configuring service. The service uses the embedded service-specific knowledge to automatically determine that a combination of an MPEG-2 server and an MPEG-2-to-MPEG-4 transcoder can provide the best video quality for the user. Also according to the service-specific knowledge, the service automatically selects a server and a transcoder to minimize the bitrate-weighted network distance for lower network resource consumption (see Figure 1). Of course, performing this composition requires a *support infrastructure* that enables service/resource discovery (e.g., finding video servers and transcoders), network measurement (e.g., determining which transcoder is closest to a video server), and component deployment (e.g., authentication, run-time execution, etc.). In this paper, we use existing solutions for the support infrastructure as described in Section 4.4.

Now let us look at two more sophisticated examples of dynamic service composition. Figure 2 shows an interactive search service that supports application-specific filtering (similar to Diamond [12]). A user wants to find a particular picture from an image collection distributed across three storage servers. For efficiency, this service allows the user to upload application-specific filters (e.g., "blue with water-like texture") to servers so that irrelevant pictures can be discarded early. The service developer determines that

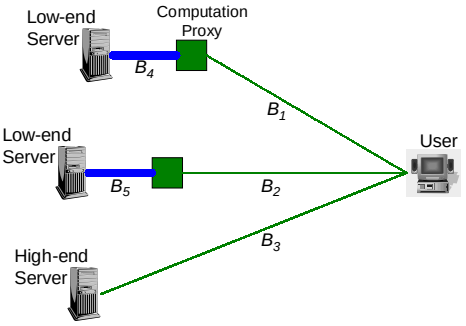


Figure 2. An interactive search service

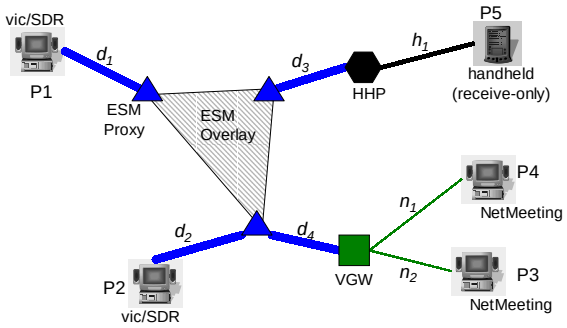


Figure 3. A video conferencing service

if a storage server does not have sufficient computation resources, a computation proxy can be used to run the filters. Furthermore, to reduce the communication time, the selection of computation proxies is based on the objective shown.

Another example is the video conferencing service in Figure 3. Five users want to hold a video conference: P1 and P2 have MBone conferencing applications vic/SDR (VIC), P3 and P4 use NetMeeting (NM), and P5 uses a receive-only handheld device (HH). The service developer's service-specific knowledge indicates that a configuration supporting these users can be composed as follows. A video conferencing gateway (VGW) is needed for protocol translation and video forwarding between VIC and NM. A handheld proxy (HHP) is needed to join the conference for P5. The service uses an End System Multicast (ESM) [2] overlay consisting of three ESM proxies (ESMPs) to enable wide-area multicast among P1, P2, VGW, and HHP, The criterion for selecting all the components is to minimize the shown objective function to reduce the network resource usage. The weights in the function reflect the bandwidth usage, e.g., NetMeeting only receives one video stream.

These examples show that service-specific knowledge is important for dynamic service composition. In this paper, we present an architecture that enables service developers to easily build self-configuring services that utilize the developers' service-specific knowledge.

3. Related work

Recently, self-configuration is identified as one of the fundamental features of "autonomic computing systems" [14, 7]. Self-configuring services in this paper more specifically refer to services that perform dynamic service composition at initialization time to satisfy each user request and system characteristics. Such services are made possible in part by the emerging Grid environments where infrastructures such as Globus [4] and the Open Grid Services Architecture [5] enable dynamic discovery, monitoring, and deployment of distributed and heterogeneous resources and service components. On the other hand, distributed component framework (e.g., CORBA [18]) and Web service technologies (e.g., SOAP [24] and WSDL [25]) simplify dynamic composition of components to provide user-level services.

To enable service developers to build self-configuring services, there are two conflicting goals: *simplicity* and *flexibility*. Previous efforts can be divided into *generic* and *service-specific* approaches, each of which addresses one of the goals. The service-specific architectures are designed for particular services or applications, for example, allocating resources dynamically for resource-intensive Grid applications [3, 15], using shortest path algorithms to select components of a given service path in peer-to-peer Grids [9], selecting components and resources given a set of multi-fidelity applications [19], and selecting resources for path-based resource-intensive applications [16]. This approach requires each service developer to implement its specialized service composition module that communicates with support infrastructures, formulates the component selection optimization problem, and solves the problem using algorithms. This approach is flexible since the developer has complete control of the composition module, However, implementing such specialized modules can be difficult for developers because it requires more than the service-specific knowledge, e.g., a developer needs to know how to formulate and solve optimization problems.

On the other hand, generic architectures can compose different types of services using "type-based composition" [8, 23, 20, 6, 13], i.e., components have well-defined input/output (requires/provides) interfaces. In the video streaming example, the user requests an MPEG-4 input. A generic service composition module can automatically determine that this can be satisfied by a combination of an MPEG-2 server (MPEG-2-Out) and a transcoder (MPEG-2-In/MPEG-4-Out). This approach simplifies a developer's job. However, it does not (fully) utilize a developer's service-specific knowledge, which can reduce search space and provide service-specific optimization criteria.

4. Recipe-based service composition

In this section, we present our architecture for recipe-based service composition and describe how service-specific knowledge can be represented in recipes.

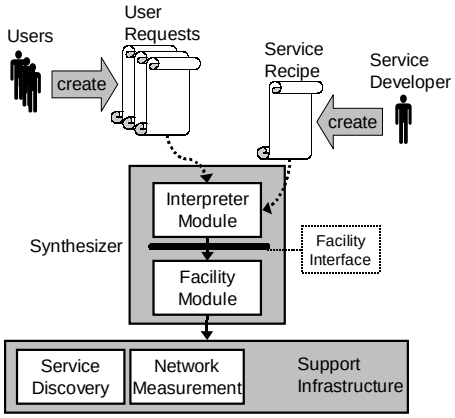


Figure 4. An architecture for recipe-based service composition

4.1. Architecture

To achieve both flexibility and simplicity, we propose a general architecture for recipe-based service composition, shown in Figure 4. A *service recipe* is written by a service developer and contains an operational description of the service-specific knowledge. A synthesizer is generic across different services and consists of two modules. The *facility module* provides common, reusable service composition functionalities, e.g., component selection algorithms and mechanisms for accessing the support infrastructure. The *facility interface* exports the facility module functionalities in the forms of an API and libraries that can be used by developers to write recipes. The *interpreter module* executes a recipe submitted by a developer to compose a service configuration for each user request.

Note that the generic and service-specific approaches discussed earlier roughly represent two extreme cases of this architecture. In the generic approach, the "recipe" is an abstract specification containing only the highest-level service-specific knowledge, e.g., the required input type. All other tasks are automatically performed by the generic "synthesizer". This greatly limits the amount and type of knowledge that can be represented. On the other hand, the "recipe" in the service-specific approach is in fact a specialized service composition module implemented by the service developer. The "synthesizer" becomes simply an interface to the support infrastructure. This requires extra implementation efforts from the developers in addition to representing the service-specific knowledge.

In contrast, our approach lets developers design recipes using a programming language such as Java, allowing a more flexible representation of service-specific knowledge. In addition, the common service composition functionalities are provided by the synthesizer so that developers can access these functionalities in their recipes without dealing

with the implementation details. The synthesizer can take the form of a toolkit/library that can be used by developers to build self-configuring services, or a standalone entity whereby a developer builds a self-configuring service by submitting a recipe to the synthesizer.

At run time, a user can send a request to the synthesizer specifying the user requirements. In the video conferencing example (Figure 3), the user request specifies the IP addresses and conferencing applications of the five participants. When the synthesizer receives the request, it extracts the user requirements, and the interpreter module executes the recipe to perform service composition. We divide the composition process into two steps (similar to previous work [8, 23, 20, 6, 13, 9]): *abstract mapping* and *physical mapping*. In abstract mapping, the synthesizer generates an *abstract configuration* specifying what types of components (e.g., VGW, HHP, ESMP) are needed to satisfy a request. Then a *physical configuration* is generated that maps each *abstract component* in the abstract configuration to a *physical component* (e.g., VGW⇒192.168.1.1, HHP⇒192.168.2.2, etc.). Service-specific knowledge is very important in both steps. Next, we discuss how we use recipes to capture these two aspects of service-specific knowledge in a general way.

4.2. Abstract mapping knowledge

An abstract configuration is a graph consisting of nodes representing the abstract components and links representing the connections between the components. While the synthesizer needs to use the knowledge in the recipe to decide what components and connections to use, the data structures for the graph are generic and can be reused across services, so the service developer should not have to define service-specific data structures to handle abstract configurations. Therefore, in our approach, a developer uses generic data structures and functions implemented by the facility module to specify the knowledge of how to construct abstract configurations in its recipe. Figure 5 lists the major data structures and functions that are made available to developers through the facility interface.

Using the video conferencing service as an example, Figure 6 shows how the service-specific abstract mapping knowledge can be expressed in the recipe using this interface. The right hand side is the recipe, basically a program written in Java. We divide the recipe into three segments. The left hand side illustrates what service-specific knowledge each of the recipe segments represents. The unshaded lines in the recipe correspond to the abstract mapping knowledge, and the shaded lines are for physical mapping. Here we look at how the abstract mapping knowledge is represented in the three recipe segments. The first segment inserts a VGW into the abstract configuration and connects it with every participant who uses NetMeeting.

Data structure

`AbsConf`	Represent abstract configurations; contains components and connections.
`AbsComp`	Represent abstract components; contains component properties (also sub-components, if any).
`PhyComp`	Represent physical components (i.e., fixed nodes such as the users); contains component properties.

Function

`addComp(spec)`	Add an abstract component with the given specifications `spec` to the abstract configuration.
`addConn(c1,c2)`	Add a connection between components c1 and c2 to the abstract configuration.
`addSubComp(n,spec)`	Add *n* identical sub-components (with `spec`) to a component in the abstract configuration.
`getProperty(prop)`	Get the value of the property named `prop` of a component in the abstract configuration.

Figure 5. Abstract configuration API

The second segment adds an HHP for each handheld participant and connects them. The third segment inserts an ESM component with three ESMP sub-components and connects ESM with the multicast endpoints. Note that `participants` is the list of participants extracted from a user request by the synthesizer; for simplicity, we use symbolic names (e.g., "VGW") to represent complete specifications of component type and attributes (e.g., "(serviceType = VideoGateway) (protocols = H323,SIP)...").

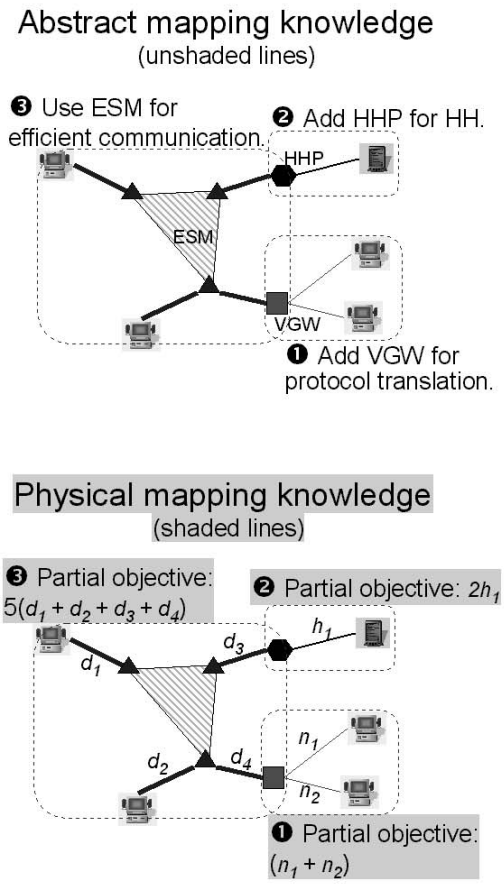
Given this recipe, generating an abstract configuration is straightforward. When a user request is received, the synthesizer executes the recipe to construct the abstract configuration `conf`.

4.3. Physical mapping knowledge

Given an abstract configuration, the synthesizer needs to generate a physical configuration specifying which physical component should be selected for each abstract component such that an *objective* is optimized. Therefore, physical mapping involves identifying the objective function, formulating the optimization problem, and solving the problem.

An important observation is that while the optimization objective is service-specific, the latter two tasks are relatively generic, e.g., optimization algorithms can be reused for different services. Moreover, developers may not have the expertise to formulate and solve optimization problems. Therefore, our approach is to let a service developer specify a service-specific *objective function* in the recipe using the facility interface. The facility module then formulates an optimization problem accordingly and solves it using built-in algorithms.

An objective function is a function of *metric terms*, e.g., "5*latency(server,transcoder) + latency(transcoder,user)". To provide an interface for specifying such functions, we first need to decide what metrics can be used in objective functions. Metrics represent properties of components or properties of connections between components. To use a metric in objective functions, the synthesizer must be able to obtain the represented property from the support infrastructure. For example, our prototype support infrastructure pro-




```
❶
AbsConf conf = new AbsConf();
Term obj = new Term(new Double(0.0));
Term partialObj = new Term(new Double(0.0));
AbsComp vgw = conf.addComp("VGW");
for (i = 0; i < participants.size(); i++) {
  PhyComp pc = (PhyComp) participants.get(i);
  if (pc.getProperty("App").equals("NM")) {
    conf.addConn(vgw, pc);
    partialObj.add(new Term(new LatencyM(vgw, pc)));
  }
}
obj.add(partialObj);

❷
List hhps = new ArrayList();
partialObj = new Term(new Double(0.0));
for (i = 0; i < participants.size(); i++) {
  PhyComp pc = (PhyComp) participants.get(i);
  if (pc.getProperty("App").equals("HH")) {
    AbsComp hhp = conf.addComp("HHP");
    conf.addConn(hhp, pc);
    partialObj.add(new Term(new LatencyM(hhp, pc)));
    hhps.add(hhp);
  }
}
partialObj.multiplyBy(new Term(new Double(2.0)));
obj.add(partialObj);

❸
AbsComp esm = conf.addComp("ESM");
esm.addSubComp(3, "ESMP");
partialObj = new Term(new Double(0.0));
for (i = 0; i < participants.size(); i++) {
  PhyComp pc = (PhyComp) participants.get(i);
  if (pc.getProperty("App").equals("VIC")) {
    conf.addConn(esm, pc);
    partialObj.add(new Term(new LatencyM(esm, pc)));
  }
}
for (i = 0; i < hhps.size(); i++) {
  AbsComp hhp = (AbsComp) hhps.get(i);
  conf.addConn(esm, hhp);
  partialObj.add(new Term(new LatencyM(esm, hhp)));
}
conf.addConn(esm, vgw);
partialObj.add(new Term(new LatencyM(esm, vgw)));
partialObj.multiplyBy(new Term(new Double(5.0)));
obj.add(partialObj);
Function objfunc = new Function(obj);
```

Figure 6. A video conferencing recipe

Data structure	
`LatencyM`	Represent the network latency between two components.
`BandwidthM`	Represent the available bandwidth between two components.
`Function`	Represent an objective function; contains a `Term`.
`Term`	Contain a metric or a floating point number; also provides member functions listed on the right for appending other instances of `Term` to "this" instance.

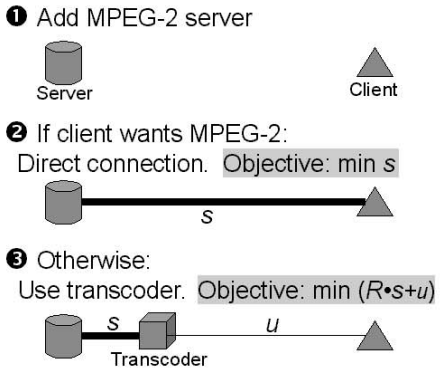
Function (member functions of `Term`)	
`add(t)`	Add `t` (an instance of `Term`) to "this" instance.
`subtract(t)`	Subtract `t` from this instance.
`multiplyBy(t)`	Multiply this instance by `t`.
`divideBy(t)`	Divide this instance by `t`.
`pow(t)`	Raise this instance to the `t`-th power.

Figure 7. Objective function API

vides network latency information, so the synthesizer can allow "latency" to be used in objective functions. Currently, we define two metrics, `LatencyM` and `BandwidthM`, for use in objective functions. Other important metrics such as CPU speed, memory size, and cost can be added by extending the support infrastructure to provide information on these properties and extending the synthesizer to accept these metrics in objective functions.

Given a set of metrics, we define the data structures and functions shown in Figure 7 for constructing objective functions. Note that since our prototype uses the Java programming language, all the data structures are defined as classes. All functions listed are member functions of the `Term` class, e.g., if `A` and `B` are both instances of `Term` and represent a and b, respectively, then after calling `A.add(B)`, `A` represents $(a+b)$. To summarize, this interface can be used to construct a tree-like data structure representing the objective function. This tree can then be traversed to evaluate the function. This API is fairly general and allows a developer to construct non-trivial objective functions in recipes.

As an example, the video conferencing developer's objective function (Figure 3) is a weighted sum of several latencies in the abstract configuration. The shaded lines in the video conferencing recipe in Figure 6 show how this objective function is constructed using the above API. On the left hand side of the figure, we illustrate which part of the objective function each segment of the recipe constructs. These partial objectives are added together in `obj`, which is finally

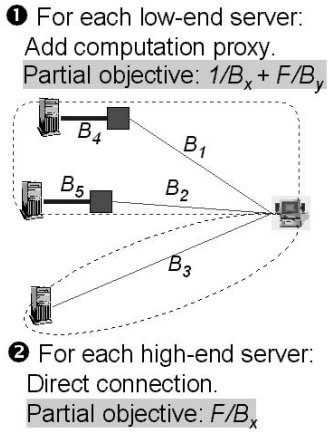



```
❶ AbsConf conf = new AbsConf();
  Term obj = new Term(new Double(0.0));
  AbsComp vserver = conf.addComp("MPEG2VideoServer");

❷ if (client.getProperty("VideoIn").equals("MPEG2")) {
    conf.addConn(vserver, client);
    obj.add(new Term(new LatencyM(vserver, client)));

❸ } else {
    AbsComp transcoder = conf.addComp("Transcoder");
    conf.addConn(vserver, transcoder);
    Term partialObj = new Term(new LatencyM(vserver, transcoder));
    partialObj.multiplyBy(new Term(new Double(MPEG2BitRate)));
    obj.add(partialObj);
    conf.addConn(transcoder, client);
    partialObj = new Term(new LatencyM(transcoder, client));
    partialObj.multiplyBy(new Term(new Double(MPEG4BitRate)));
    obj.add(partialObj);
  }
  Function objfunc = new Function(obj);
```

Figure 8. A video streaming recipe




```
AbsConf conf = new AbsConf();
Term obj = new Term(new Double(0.0));
for (i = 0; i < storageServers.size(); i++) {
  PhyComp sd = (PhyComp) storageServers.get(i);
  if (belowThreshold(sd.getProperty("CompResource"),
                     MinCompResource)) {
❶   AbsComp cproxy = conf.addComp("CompProxy");
    conf.addConn(sd, cproxy);
    Term partialObj = new Term(new Double(1.0));
    partialObj.divideBy(new Term(new BandwidthM(sd, cproxy)));
    obj.add(partialObj);

    conf.addConn(cproxy, client);
    partialObj = new Term(new Double(EstFilteringRatio));
    partialObj.divideBy(new Term(new BandwidthM(cproxy,client)));
    obj.add(partialObj);

❷ } else {
    conf.addConn(sd, client);
    Term partialObj = new Term(new Double(EstFilteringRatio));
    partialObj.divideBy(new Term(new BandwidthM(sd, client)));
    obj.add(partialObj);
  }
}
Function objfunc = new Function(obj);
```

Figure 9. An interactive search service recipe

used to construct the function `objfunc`.

4.4. Implementation

We have implemented the synthesizer, including the interpreter and facility modules and the interface, in Java. The service recipes are also written in Java and use the classes and interface provided by the facility module. The interpreter module dynamically loads a compiled recipe and invokes an entry function. The facility module accesses the support infrastructure to carry out actual operations. We also implemented a working self-configuring video conferencing system as depicted in Figure 3.

For the support infrastructure, we use the Global Network Positioning (GNP) [17] approach to compute the coordinates of network nodes, which are used to estimate the network latencies between nodes to support the `LatencyM` metric. Note that the `BandwidthM` metric is not supported yet since we have not integrated a bandwidth measurement infrastructure. We also designed and implemented the Network-Sensitive Service Discovery (NSSD) [11] infrastructure that provides, in addition to traditional service discovery, the capability to return the best *m* candidates given a local optimization criterion for a component.

4.5. Expressiveness

To evaluate the expressiveness of the recipe representation, we apply the recipe representation to the other two services in Section 2 in addition to video conferencing. Figure 8 shows a recipe for the video streaming service (Figure 1), and Figure 9 is a recipe for the interactive search service (Figure 2). The video streaming recipe constructs different abstract solutions depending on whether the user is requesting MPEG-2 or MPEG-4 video. The user is represented by the physical component `client` extracted from the user request. The objective function in Figure 1 is easily constructed using the objective function API.

In the interactive search service recipe, `client` (the user) and `storageServers` (the storage servers) are fixed physical components extracted from the user request. For each server, if it does not have sufficient computation resources, the recipe adds a computation proxy and its "contribution" to the global objective in Figure 2. In these examples, the recipe representation allows us to express the service-specific knowledge easily and flexibly.

Of course, developers building the same service may design different recipes. For example, instead of the recipe in Figure 8, a developer of a video streaming service may come up with the simpler recipe in Figure 10 that constructs an objective function involving only local optimization. Using this "local" recipe, the synthesizer may not produce the globally optimal configuration, but the synthesizer will spend significantly less time in configuring the service.

To see the effect of the different video streaming recipes, we perform a set of simple simulations. We obtain the GNP coordinates of 869 Internet nodes from the GNP project [17], so the latencies between the nodes are realistic. We randomly select 600 nodes to represent clients, servers, and transcoders (200 each). For each client, we find the best pair of server and transcoder using either the "global" recipe or the "local" recipe. This is repeated 20 times for a total of 4000 scenarios. We compute the resource consumption (i.e., the global objective) of the resulting configuration in each scenario and also measure the time it takes to find the configuration. The average resource consumption is 90.79 for global and 203.41 for local. The time measurements (total of 4000 scenarios) are 158.20 seconds for global and 2.05 seconds for local. This experiment clearly shows the trade-off between optimality and optimization cost in the two recipe designs. Therefore, developers can write different recipes according to their service-specific requirements.

5. Solving the physical mapping problem

After constructing the abstract configuration `conf` and the objective function `objfunc`, the recipe execution is complete. Given these results, the synthesizer then needs to solve the physical mapping problem, i.e., for each abstract component (e.g., VGW), the synthesizer needs to find the candidates (e.g., machines running the gateway software) through a service discovery infrastructure and then select the best one such that the objective function is optimized. Let us first discuss the complexity of such problems and algorithms for solving them.

5.1. Complexity and algorithms

In general, the synthesizer cannot simply select each component independently. For example, if the objective function includes a single metric $M(c_1, c_2)$ (c_1 and c_2 are abstract components with n candidates each), the selection of c_1 *depends* on that of c_2, i.e., the synthesizer needs to select them together (n^2 possible combinations) to find the optimal selection. Furthermore, this dependency is *transitive*, e.g., if the objective is $M_1(c_1, c_2) + M_2(c_2, c_3)$, then all three are mutually dependent. In addition, some physical mapping problems involve *semi-dependent* components, e.g., in $M_1(c_1, c_2) + M_2(c_2)$, c_2 is semi-dependent (independent in the second term but depends on c_1 in the first term). Similarly, VGW and HHP in Figure 3 are both semi-dependent.

```
AbsConf conf = new AbsConf();
AbsComp vserver = conf.addComp("MPEG2VideoServer");

if (client.getProperty("VideoIn").equals("MPEG2")) {
  conf.addConn(vserver, client);
} else {
  AbsComp transcoder = conf.addComp("Transcoder");
  conf.addConn(vserver, transcoder);
  conf.addConn(transcoder, client);
}
Term obj = new Term(new LatencyM(vserver, client));
Function objfunc = new Function(obj);
```

Figure 10. A simpler video streaming recipe

If some or all components are mutually dependent, physical mapping is a *global optimization* problem with a worst-case problem size n^m where m is the number of abstract components. If every abstract component is independent, e.g., if objective is $M_1(c_1) + M_2(c_2) + \ldots + M_m(c_m)$, physical mapping becomes a series of *local optimization* problems with a total problem size mn.

Previously, researchers have proposed many optimization techniques for this type of optimization problems. For example, a dynamic programming algorithm is used in CANS [6] to map components to nodes along a selected route to optimize the overall throughput. Choi et al. uses a shortest-path algorithm on a transformed network graph to select intermediate processing sites between two end points [1]. Gu and Nahrstedt uses a shortest-path algorithm to find a service path that minimizes the resource usage [9]. The Matchmaking framework [21] maps a computation task to an appropriate resource that optimizes user-specified criteria. Liu et al. [15] and Raman et al. [22] extend Matchmaking to support mapping a task to multiple resources and to support resource co-allocation, respectively, and they propose a number of heuristic algorithms to solve the mapping problem. Each of these studies addresses a subset or a particular form of the mapping problems. In contrast, since in our approach a developer can specify a broad range of objective functions, the synthesizer must be able to solve a broader range of mapping problems.

Currently, we have implemented the following more general optimization algorithms in our prototype.

- *Exhaustive search*: This algorithm always yields the actual optimal configuration, but the optimization cost grows rapidly with the problem size.
- *Sim-anneal(R)*: The simulated annealing heuristic is based on the physical process of "annealing" [10]. We use the temperature reduction ratio R as a parameter to control the cost/optimality trade-off.
- *Hybrid(m)*: This heuristic is for problems with semi-

dependent components, e.g., HHP and VGW in video conferencing. The synthesizer reduces the problem size by choosing *m* "local candidates" for each semi-dependent component according to its "independent metric", and global optimization is performed on the reduced search space. Preliminary evaluation results for this heuristic are presented in [11].

• *HybridSA(m)*: This is the same as Hybrid(*m*) except that the final global optimization is performed using simulated annealing. The temperature reduction ratio increases with *m* to achieve better optimality (at higher costs).

Note that specialized algorithms such as those developed in previous studies can also be implemented to handle special cases more efficiently.

5.2. Selecting the best algorithm

The algorithms discussed above are suitable under different circumstances. Therefore, for each physical mapping problem, the most appropriate algorithm needs to be selected. However, developers may not be optimization experts and may not know the properties of different algorithms well enough to select the best algorithm. Therefore, we propose that the synthesizer should be able to automatically choose the best optimization technique when solving a physical mapping problem. Next, we present a high-level design for automatic algorithm selection. We have not yet implemented this design in our prototype.

The best choice of optimization technique depends on two major factors. The first is the properties of the problem itself, e.g., when the problem size is small, an expensive algorithm can be used to achieve better optimality. Similarly, specialized algorithms can be used for problems of particular forms, e.g., path-based or having semi-dependent components. These properties are not service-specific and can be analyzed by the synthesizer automatically to choose an algorithm. For example, using the video conferencing recipe, the synthesizer generates an abstract configuration and an objective function. By looking at the number of candidates for each component, the synthesizer can determine the problem size and estimate the feasibility of using an expensive algorithm such as exhaustive search. In addition, the synthesizer can also analyze the objective function and discover that the mapping problem involves semi-dependent components. Therefore, special heuristic algorithms for such problems can be used.

The second factor is the desired trade-off between the optimality of the resulting configuration and the cost of optimization. The desired trade-off for each service is service-specific and is determined by the developer, e.g., some developers have a maximum optimization cost constraint and do not require a near-optimal configuration, while others are willing to spend more resources on optimization and have a stricter optimality requirement. However, as discussed above, developers may not know which algorithm can achieve the desired trade-off.

To consider a developer's service-specific trade-off when choosing an algorithm, the synthesizer can provide an interface for developers to specify in their recipes the desired cost/optimality trade-off. A simple interface for this purpose has the following two functions:

```
setAlgCostConstraint(CostSpec maxCost);
setAlgOptimalityConstraint(OptSpec minOptimality);
```

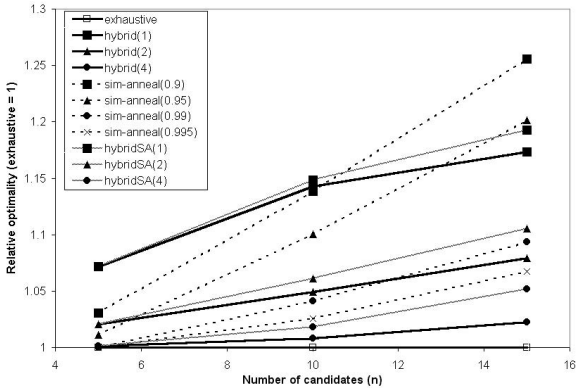
Using this interface, developers can specify two constraints in their recipes: the maximum optimization cost and the minimum optimality of the resulting configuration. Currently, we are refining this interface definition based on preliminary evaluation results and implementing this capability in the synthesizer. For example, one simple approach is to define "regions" of cost and optimality, e.g., "high" and "low" for both `CostSpec` and `OptSpec`. A more precise specification will require a definition that captures cost and optimality quantitatively.

Given these two constraints specified by the developer, the synthesizer needs to select an algorithm that satisfies the constraints for each mapping problem. Therefore, the synthesizer needs to know the relation between the cost and optimality properties of the various algorithms. Currently, we have not yet implemented the automatic algorithm selection. One possible approach is that the synthesizer can "learn" from its experiences by, for example, periodically using an expensive algorithm to sample the optimality and keeping a history of the cost/optimality relation to derive a guideline for algorithm selection. Another possibility is that through measurement or analysis, we can "pre-compute" the cost/optimality relation for the algorithms under common forms of the mapping problem and then embed this knowledge in the synthesizer as the selection guideline.

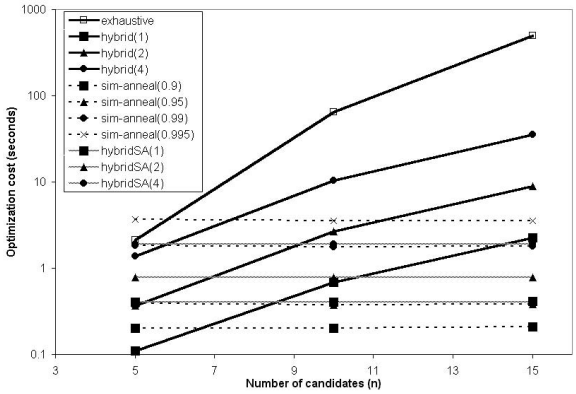
More specifically, for example, we can pre-compute a "table" of cost and optimality of each algorithm under different problem sizes. When the synthesizer needs to solve a mapping problem, the following procedure can be used to select the best algorithm. The synthesizer first analyzes the problem to determine what algorithms are eligible, and then it determines the problem size and looks up the table to retrieve the cost and optimality properties of the candidate algorithms under the particular problem size. Finally, the synthesizer examines the cost and optimality constraints specified by the developer to select the best algorithm.

5.3. Using pre-computed data

As an example, we simulate the video conferencing scenario in Figure 3 and look at how pre-computed cost and optimality data can be used by the synthesizer to select an algorithm. The problem size *n* in our simulations ranges from 5 to 200 (each of VGW, HHP, and ESMP has *n* candidates). We generate 200 requests (5 participants each) in 10 different candidate distributions. Similar to the simula-

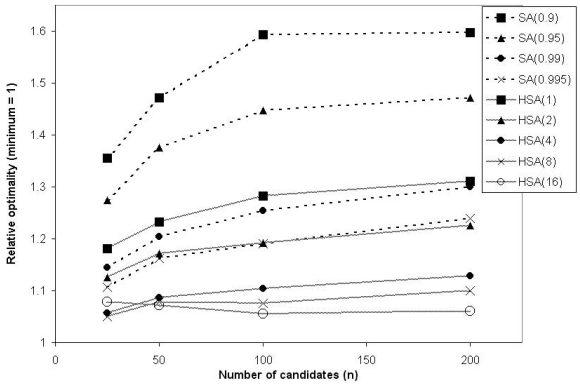


(a) Optimality of resulting configurations

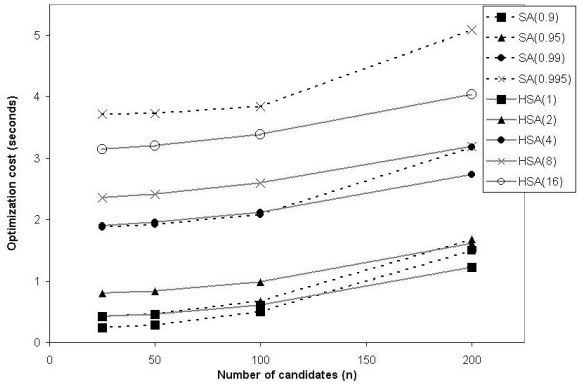


(b) Optimization cost

Figure 11. Optimization for small-scale problems



(a) Optimality of resulting configurations



(b) Optimization cost

Figure 12. Optimization for larger-scale problems

tions earlier, the candidates and participants are randomly selected from the set of 869 nodes with GNP coordinates.

For each request, the synthesizer executes the recipe (Figure 6) and uses the implemented algorithms to find the optimal configuration. We compute the average *relative optimality* (RO) of each algorithm, e.g., if an algorithm has RO 1.5, on average the generated configuration is 50% worse than the actual optimum. We also measure the average *optimization time per request* to represent the optimization cost. Given that the optimization time is measured using our unoptimized Java prototype running on an average desktop machine (Pentium III 933MHz CPU, Red Hat Linux 7.1, and J2SE 1.4.2), the measurements should only be used for comparing different techniques in our simulations. We expect to see much better performance using an optimized C/C++ implementation in a production environment.

Figure 11(a) shows the RO of different algorithms for n between 5 and 15, and Figure 11(b) shows the optimization cost. Exhaustive search and Hybrid offer good optimality, but their costs increase rapidly with n ($O(n^5)$ for exhaustive; $O(m^2n^3)$ for Hybrid). Simulated annealing (SA) and HybridSA have almost constant costs, but their optimality deteriorates faster than Hybrid.

For n between 25 and 200, it is no longer feasible to run exhaustive search and Hybrid, so we cannot normalize all results to the exhaustive search result. Therefore, the RO of each algorithm for each request is computed by normalizing to the best-performing algorithm for that request. Figure 12(a) shows the average RO computed this way, and Figure 12(b) shows the optimization cost. HybridSA generally performs better than SA, and the costs of both increase slowly with n (with SA growing faster due to larger search spaces). Note the fact that the curves become flat for n between 100 and 200 only means that all algorithms deteriorate at roughly the same rate since we normalize to the best-performing algorithm.

These results can be used as guidelines for the synthesizer to select the appropriate algorithm for each instance of this mapping problem based on both the specified cost/optimality trade-off and the properties of the problem. For example, assume that for a given request, the resulting physical mapping problem size is small ($n = 10$), the cost

constraint is 100, and the optimality constraint is 1.01 (i.e., at most 1% worse than actual optimum). The synthesizer can look at the results for $n = 10$ and find that only the exhaustive algorithm satisfies both constraints. Similarly, if the problem size is large ($n = 100$), cost constraint is 0.75, and optimality constraint is 1.3, then the synthesizer will select HybridSA(1).

To summarize, by providing an interface for developers to specify their desired cost/optimality trade-off, our approach allows developers to customize algorithm selection without knowing the details of the algorithms.

6. Conclusion

We presented a general architecture for service developers to build self-configuring services that can use distributed components to dynamically compose an optimal service configuration according to particular user requirements and system characteristics. We design a recipe representation that can be used by developers to easily capture their service-specific knowledge, and we develop a synthesizer that performs dynamic service composition automatically according to the knowledge in a recipe. We apply this architecture to three different services to demonstrate its flexibility and simplicity. Finally, based on simulation results, we derive guidelines for the synthesizer to select an appropriate optimization algorithm according to a developer's service-specific trade-off between optimality of component selection and optimization cost.

References

[1] S. Choi, J. Turner, and T. Wolf. Configuring Sessions in Programmable Networks. In *Proceedings of IEEE INFOCOM 2001*, Apr. 2001.

[2] Y. Chu, S. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM Sigmetrics*, June 2000.

[3] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. *IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[4] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[5] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35(6), June 2002.

[6] X. Fu, W. Shia, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. In *Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS '01)*, Mar. 2001.

[7] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[8] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *IEEE Computer Networks, Special Issue on Pervasive Computing*, 35(4), Mar. 2001.

[9] X. Gu and K. Nahrstedt. A Scalable QoS-Aware Service Aggregation Model for Peer-to-Peer Computing Grids. In *The 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[10] J. Hromkovič. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer, 2001.

[11] A.-C. Huang and P. Steenkiste. Network-Sensitive Service Discovery. In *The Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Mar. 2003.

[12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. Ganger, E. Riedel, and A. Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. IRP-TR-03-09, Intel Research, Oct. 2003.

[13] A.-A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, 2002.

[14] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[15] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and Evaluation of a Resource Selection Framework for Grid Applications. In *IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 2002.

[16] J. López and D. O'Hallaron. Evaluation of a resource selection mechanism for complex network services. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing*, Aug. 2001.

[17] T. S. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proceedings of IEEE INFOCOM 2002*, June 2002.

[18] Object Management Group. CORBA: The Common Object Request Broker Architecture, Revision 2.0, July 1995.

[19] V. Poladian, D. Garlan, M. Shaw, and J. P. Sousa. Dynamic Configuration of Resource-Aware Services. In *Proceedings of the 26th International Conference on Software Engineering*, May 2004.

[20] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. *The Eleventh World Wide Web Conference (Web Engineering Track)*, May 2002.

[21] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *The Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[22] R. Raman, M. Livny, and M. Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proceedings of the Twelfth IEEE International Symposium on High-Performance Distributed Computing*, June 2003.

[23] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. Automated Planning for Open Architectures. In *OPENARCH 2000 – Short Paper Session*, pages 17–20, Mar. 2000.

[24] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[25] Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.