Distributed Automatic Service Composition in Large-Scale Systems

Songlin Hu[⋆], Vinod Muthusamy[†], Guoli Li[‡], and Hans-Arno Jacobsen[†‡]
husonglin@ict.ac.cn, {vinod@eecg, gli@cs, jacobsen@eecg}.toronto.edu

[⋆]Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[†]Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
[‡]Dept. of Computer Science, University of Toronto, Toronto, Canada

ABSTRACT

Automatic service composition is an active research area in the field of service computing. This paper presents a distributed approach to automatically discover a composition of services based on the desired input to and output from the process. The algorithm makes use of the content-based publish/subscribe model, with service inputs modeled as subscriptions, and outputs as advertisements. Service interfaces are mapped to publish/subscribe messages in such a way that publish/subscribe matching is used to evaluate service compatibility. In this way, large-scale distributed service composition and process discovery is achieved with a distributed publish/subscribe network. Evaluations in a distributed environment of a real implementation of the system demonstrate the scalability of the distributed approach, especially with respect to the number of services, the complexity of the discovered processes, and the number of concurrent searches.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Automatic service composition, process discovery, publish/subscribe, multi-agent systems, routing, distributed system

1. INTRODUCTION

Service computing is a computing paradigm that is widely used to realize distributed applications based on loosely coupled, reusable services [1, 7, 19]. There is much industry support for service composition as evidenced by the popularity of service oriented architecture (SOA) platforms, and process execution languages such as BPEL and DAML-S. However, the composition of services into a process is still typically a labor-intensive task, only made worse with wider adoption of service computing and proliferation of the number of available services that may be composed. What is needed is a way for the user to search for processes, or service compositions, that meet some functional criteria, allowing the user to then examine the results and select appropriate processes for direct use or further tuning. This problem is known as *automatic service composition*, and has been an active field of research [3, 4, 9, 11, 14, 16, 17, 18, 21, 23] and has garnered interest in industry [15, 20].

With automatic service composition, a repository of available services is searched, and compositions of services that meet some user defined criteria are found. Often the criteria indicates some input and output requirements. For example, a user wishing to print an HTML page, may search for a process that accepts as input an HTML document, and returns as output the status of the print job. However, it may be that the printer service, W_p, only accepts a PostScript file as input, but there is another service, W_c, that can convert an HTML document into a PostScript file. In this case, the search result would be a process consisting of the two services W_p and W_c executed in series. A more complex example is a property developer who wants to obtain a permit to build a condominium in a city. While this is a common process, the developer may not be familiar with the city's laws, and does not want to manually examine the many Web Services offered by the municipality. Instead, the developer searches for a composition of services whose input is a request for a permit, and whose output is a building permit. A process may be found that composes services to first obtain zoning clearances, followed by an environmental assessment, and only then request electricity, water, and sewage approvals (perhaps in parallel). Finally, a service collects the approvals from the various utilities, and returns a building permit to the user.

Many algorithms have been proposed to solve the problem of automatic service composition, or process search, using techniques such as AI planning and theorem proving [16]. Most of them are able to discover a sequential composition of services by searching the repositories and selecting successors recursively.

This paper proposes using a publish/subscribe model [5, 6, 8] to solve the process search problem. Pub/Sub is a

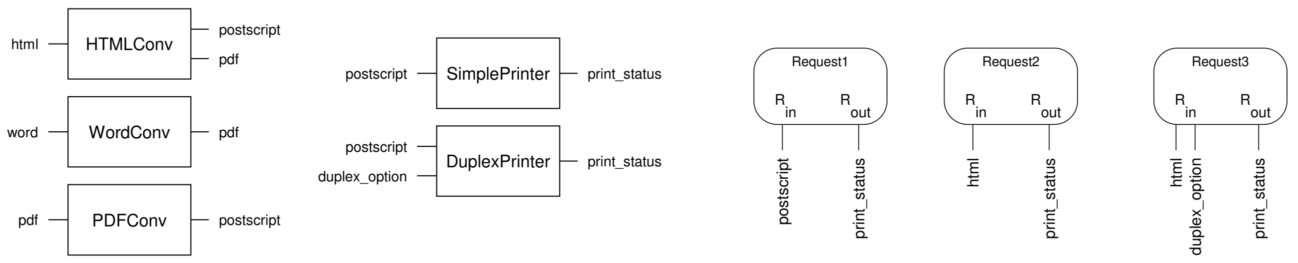


Figure 1: Example services and requests

messaging paradigm in which one or more brokers support decoupled interaction among information sources and sinks. It turns out that determining the interaction relationships among the data senders and receivers in a pub/sub system bears some similarity to discovering possible service compositions. Therefore, with an appropriate mapping of the problem space, the data structures and algorithms used by a pub/sub broker for pub/sub matching can be exploited for process search. This is a unique use of a pub/sub infrastructure and allows us to take advantage of existing work on efficient and scalable pub/sub algorithms.

To the best our knowledge, existing automatic service composition algorithms are centralized, and may have difficulty scaling to large service repository sizes, many concurrent searches, or distributed repositories. Distributed process search seems to be a more appropriate architecture since service computing systems are typically already distributed, with services scatted geographically, administered by disparate administrative domains, and registered with several autonomous registries. Moreover, distributed search has the potential to address certain shortcomings of centralized algorithms, including scalability, parallelism, and the ability to more efficiently utilize the resources available in a distributed environment.

This paper presents a distributed architecture for automatic service composition. By observing the similarity between finding compatible services in a process and pub/sub matching, and representing the former problem as the latter, the distributed process search is optimized using state of the art distributed pub/sub matching and routing techniques.

The key contributions of the paper are as follows.

1. The problem of automatic service composition is addressed by modeling services using content-based pub/sub messages. A mapping from one problem domain to the other is done in a way that the matching of pub/sub messages also indicates service composition relationships.

2. A distributed process search algorithm is developed that uses the relationships discovered above to compose services into a process that satisfies the user's input and output interface criteria.

3. An evaluation and detailed analysis of the algorithm is performed using a real implementation in a distributed environment. The sensitivity of the distributed process search algorithm compared to a centralized one is investigated with respect to various factors.

The paper continues with Section 2 introducing some terminology and background on the automatic service composition problem and the pub/sub model. Then, Section 3 presents a distributed process search architecture including the mapping of the problem to pub/sub terms and an algorithm to realize distributed, parallel process search. An evaluation of the algorithm in analyzed in Section 4. Section 5 puts this work in the context of related research in the area, with concluding remarks outlined in Section 6.

2. BACKGROUND

This section presents the basic service composition problem and some introductory content-based pub/sub concepts.

2.1 Service Composition

A *service* W is defined by input and output interfaces, (W_{in}, W_{out}), respectively. Each interface, in turn, consists of a set of *parameters*: $W_{in} = \{I_1, \ldots, I_m\}(m >= 1)$ and $W_{out} = \{O_1, \ldots, O_n\}(n >= 1)$. For a WSDL Web Service, an interface may correspond to an individual operation, and the parameters of the input and output interfaces may correspond to the input and output message schemas of an operation. Likewise, for a service invoked by an RPC mechanism such as Java RMI, the set of argument types of a method would map to the input interface parameters, and the method return type to the output interface parameter. Figure 1 shows a few services with different number of input and output parameters. For example, the `DuplexPrinter` service has two input parameters of type `postscript` and `duplex_option`, and one output parameter that is of type `print_status`.

An invocation R of a service is defined by the request parameters $R_{in} = \{I_1, \ldots, I_p\}(p >= 1)$ and expected response parameters $R_{out} = \{O_1, \ldots, O_q\}(q >= 1)$. Figure 1 illustrates three requests. For example, `Request3` has two input parameters of type `html` and `duplex_option`, and one output parameter of type `print_status`.

The sections below define some additional terms.

2.1.1 Successor

In a composition of services, a service W' may *succeed* another service W if the expected input parameters of W' are contained within the set of output parameters from W: $W_{out} \supseteq W'_{in}$. Equivalently, it can be said that W *precedes* W'.

This definition is extended to include the invocation R, such that service W succeeds R if $R_{in} \supseteq W_{in}$, and R succeeds W if $W_{out} \supseteq R_{out}$.

For example, for the services from Figure 1, service `SimplePrinter` succeeds service `PDFConv`, and `Request1` precedes service `SimplePrinter`.

2.1.2 Compatibility

Two services are said to be compatible if some of the input parameters of one is available as an output parameter of another. Formally, W is compatible with W' if $W_{out} \cap W'_{in} \neq \emptyset$. Likewise, an invocation R is compatible with service W if $R_{in} \cap W_{in} \neq \emptyset$ or $R_{out} \cap W_{out} \neq \emptyset$. Notice that the definition of compatibility is weaker than that of successors, and that a successor relationship implies compatibility.

From the examples in Figure 1, service `PDFConv` is compatible with (and precedes) service `SimplePrinter`, whereas the `HTMLConv` service is compatible with (but does not precede) service `DuplexPrinter`.

2.1.3 Atomic Process

It is possible that a single service W can fully satisfy the input and output requirements of an invocation R. In this case, the "process" that fulfills R is an *atomic* process consisting of the single service W.

Formally, W satisfies R if and only if

- $R_{in} \supseteq W_{in}$, and
- $W_{out} \supseteq R_{out}$.

That is, R_{in} precedes W which precedes R_{out}.

An atomic process that fulfills `Request1` would consist of the single service `SimplePrinter`.

2.1.4 Composite Process

It is probably more common that a composition of services is required to satisfy an invocation's input and output requirements.

A composition of services is represented as a directed cyclic graph G in which the nodes are services, and edges denote compatibility relationships. More precisely, a directed edge from W to W' means that $W_{out} \cap W'_{in} \neq \emptyset$.

The directed graph in Figure 2 illustrates all the possible compatibility relationships among the services in Figure 1 and `Request3`.

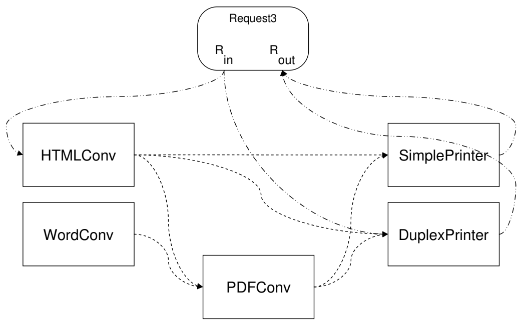


Figure 2: Compatibility graph

The compatibility constraints of the above graph G are not sufficient for the process defined by the graph to fulfill an invocation's requirements. Instead, what is required is a subset of G that is acyclic and where every service's input parameters are fully satisfied by one or more predecessors. Formally, an invocation R is fulfilled by a composite process represented by a directed acyclic graph (DAG) D consisting of a sequence of services $\{W1, W2, \ldots, Wx\}(x >= 2)$ if and only if

- R_{in} precedes $W1_{in}$,
- $(R_{in} \cup W1_{out} \cup \ldots \cup Wi - 1_{out}) \supseteq Wi_{in}(1 < i \leq x)$, and
- $(R_{in} \cup W1_{out} \cup \cdots \cup Wx_{out}) \supseteq R_{out}$.

There may be more than one DAG that fulfills an invocation request. For example, Figure 3 illustrates the two DAGs that result from the compatibility graph in Figure 2. In the first DAG in Figure 3, note that the `DuplexPrinter` service gets one of its inputs directly from the request, and another from the `HTMLConv` service. In the second DAG in Figure 3, the `pdf` output from the `HTMLConv` service is passed to the `PDFConv` service whose `postscript` output is then input to the `DuplexPrinter` service. While both processes in Figure 3 satisfy `Request3`, in this case, a user will most likely favor the first since it avoids an unnecessary invocation of the `PDFConv` service.

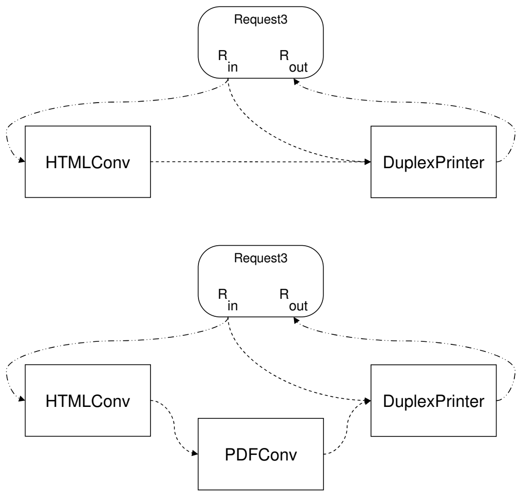


Figure 3: DAGs resulting from Figure 2

The two DAGs in Figure 3 are the results of the search for a process for invocation `Request3`. These two processes are returned to the user, who will decide which process to use. In this case, the first one is preferable since it avoids an unnecessary use of the `PDFConv` service.

2.2 Content-based pub/sub

Pub/Sub is a push-based messaging paradigm that supports decoupled and anonymous interaction between publishers that produce data in the form of publications, and subscribers who register their interests by issuing subscriptions. One dimension along which to compare pub/sub systems is the expressiveness of their subscription language. In channel-based pub/sub, publications are sent to a predefined channel, and subscribers interested in a channel are notified of all publications on that channel. The approach in this paper utilizes a more powerful content-based model in which subscriptions can specify constraints on the content of publications.

This paper assumes a content-based pub/sub language based on predicate expressions and attribute-value pairs [5]. Exploiting more expressive languages such as XPATH expressions on XML documents [12] or location-based constraints [22] is left for future work.

Subscription s	Advertisement a	Intersection Relation
(product = “computer”, brand = “IBM”, price $\leq$ 1600)	(product = “computer”, brand = “IBM”, price $\leq$ 1500)	a intersects s
(product = “computer”, price $\leq$ 1600)	(product = “computer”, brand = “IBM”, price $\leq$ 1600)	a intersects s
(product = “computer”, brand = “IBM”, price $\leq$ 1600)	(product = “computer”, brand = “Dell”, price $\leq$ 1500)	a does not intersect s

Table 1: Examples of subscriptions, advertisements and intersection relations

Formally, a publication is defined as $\{(a_1, val_1), (a_2, val_2), \ldots, (a_n, val_n)\}$. Subscriptions are expressed as conjunctions of Boolean predicates of the form (*attribute_name relational_operator value*). A predicate (*a rel_op val*) matches an attribute-value pair (a', val') if and only if the attribute names are identical $(a = a')$ and the (*a rel_op val*) Boolean relation is true for value val'. A subscription s is matched by a publication p if and only if all its predicates are matched by some pair in p.

Some systems use advertisements to let publishers announce the set of publications they are going to publish. Both subscriptions and advertisements have the same representation, but the predicates in a subscription are conjunctive, while those in an advertisement are disjunctive.

An advertisement a *matches* a publication e if and only if all attribute-value pairs match some predicates in the advertisement. Formally, an advertisement $a = \{p_1, p_2, \ldots, p_n\}$ determines a publication e, if and only if $\forall(attr, val) \in e$, there exists a predicate $p_k \in a$ where $(attr, val)$ matches p_k.

An advertisement a *intersects* a subscription/advertisement s if and only if the intersection of the set of the publications determined by the advertisement a and the set of the publications that match s is a non-empty set. Formally, at the predicate level, an advertisement $a = \{a_1, a_2, \ldots, a_n\}$ intersects a subscription/advertisement $s = \{s_1, s_2, \ldots, s_n\}$ if and only if $\forall s_k \in s, \exists a_j \in a$ and there exists some attribute-value pair $(attr, val)$[1] such that $(attr, val)$ matches both s_k and a_j. Table 1 presents some examples of subscriptions and advertisements and the corresponding intersection relations.

In a distributed content-based pub/sub system [5, 6, 8], advertisements are flooded through the overlay network of pub/sub brokers forming a spanning tree rooted at the publisher. Subscriptions that intersect the advertisement are routed toward the root of the advertisement tree, forming a tree rooted at the subscriber whose leaves terminate at publishers with potentially interesting data. Finally, publications that match subscriptions are routed back up the subscription trees and delivered to interested subscribers.

3. DISTRIBUTED SERVICE COMPOSITION

This paper presents a distributed architecture for automatic service composition consisting of *service agents* and *request agents* connected to a distributed content-based pub/sub broker network as illustrated in Figure 4. Services are assigned to *service agents* which register the service by translating service input and output interfaces into pub/sub advertisement and subscription messages. *Request agents*, on the other hand, manage the requests and results of process searches. Both service and request agents connect to the broker network as ordinary pub/sub clients and do not need to be concerned with the internal details of the pub/sub matching and routing algorithms.

[1] s_k and a_j refer to the same attribute $attr$.

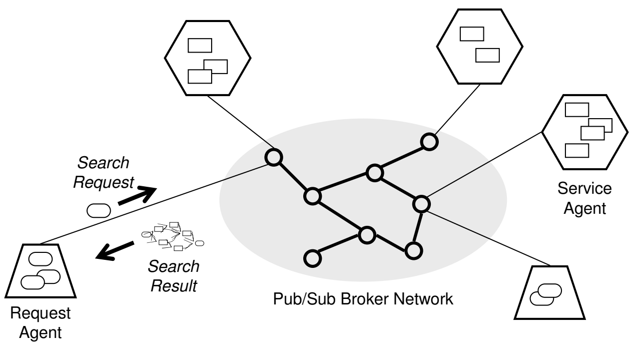


Figure 4: System architecture

The system and algorithm are fully distributed with no centralized data structures or control nodes. Service and request agents can connect to arbitrary brokers, and brokers and agents can be added or removed at any time. Furthermore, the decoupling properties of the pub/sub network allow the agents to coordinate a distributed process search among themselves without being aware of one another.

Section 3.1 describes how the broker network indirectly establishes the compatibilities among registered services, and Section 3.2 presents a distributed algorithm whereby the service agents collaborate to discover service compositions that satisfy a user's search request.

3.1 Mapping to Pub/Sub model

This section outlines how the service composition problem can be expressed in terms of content-based pub/sub messages.

Recall that a service W is defined by its input and output interfaces, $W_{in} = \{I_1, \ldots, I_m\}$ and $W_{out} = \{O_1, \ldots, O_n\}$, respectively. The input interface is mapped to a set of subscriptions and the output interface into an advertisement, such that a service is now defined as $W = \{W_{subs}, W_{adv}\}$.

W_{subs} is a set of m subscriptions, each of which contains a single predicate corresponding to a parameter in the input interface: $W_{subs} = \{s_1, \ldots, s_m\}$ $(1 \leq i \leq m)$ where s_i corresponds to I_i. The representation of a parameter by a subscription will depend on the expressiveness of the subscription language. For a predicate-based language that supports string comparisons as described in Section 2, the subscription may simply specify a globally unique string representation of the parameter type. For a WSDL Web Service, this may be the URI associated with the input message schema definition of an operation. For a Java RMI method, this may be the fully qualified class name of the method's argument types.

Notice that the input interface is mapped to a set of subscriptions each with one predicate instead of a single subscription with a conjunction of predicates. This is required so that the input to a service can be matched by the outputs

Service or request	Subscriptions	Advertisement
`SimplePrinter`	s_0 : (postscript = "*")	a_0 : (print_status >= 0)
`DuplexPrinter`	s_1 : (postscript = "*") s_2 : (duplex_option >= 0)	a_1 : (print_status >= 0)
`HTMLConv`	s_3 : (html = "*")	a_3 : (postscript = "*", pdf = "*")
`WordConv`	s_4 : (word = "*")	a_4 : (postscript = "*")
`Request1`	s_5 : (print_status >= 0)	a_5 : (postscript = "*")
`Request3`	s_6 : (print_status >= 0)	a_6 : (html = "*", duplex_option = 0)

Table 2: Subscriptions and advertisements for services in Figure 1

.

of a set of services instead of a single service.

The output interface of service W is mapped to a single advertisement W_{adv} with n predicates, each corresponding to a parameter in the output interface: $W_{adv} = \{O_1, \ldots, O_n\}$. As with the subscriptions, the representation of an output interface parameter as predicates in an advertisement will depend on the advertisement language supported by the pub/sub system.

For an invocation $R = \{R_{in}, R_{out}\}$, the input is mapped to an advertisement, and the expected output to a subscription in a similar manner to the service interfaces: $R = \{R_{adv}, R_{subs}\}$, where $R_{subs} = \{s_1, \ldots, s_q\}$.

Table 2 lists the subscriptions and advertisements issued for each service in Figure 1. Notice that the two input parameters of service `DuplexPrinter` are mapped to two subscriptions with one predicate each, but the two output parameters of the `HTMLConv` service are represented by one advertisement with predicates.

The sections below continue the mapping of the problem by redefining the terms from Section 2.1 in terms of pub/sub primitives.

3.1.1 Successor

Service W' succeeds service W if every subscription in W'_{subs} intersects W_{adv}: $\forall i \in \{1, \ldots, m\}, s_i \bowtie W_{adv}$, where $\bowtie$ is the intersection operator between subscriptions and advertisements.

For example, in Table 2, `SimplePrinter` succeeds `WordConv` because every subscription by `SimplePrinter` (s_0 in this case) intersects `WordConv`'s advertisement a_4.

3.1.2 Compatibility

Service W is compatible with service W' if at least one subscription in W_{subs} intersects W_{adv}: $\exists i \in \{1, \ldots, m\}, s_i \bowtie W_{adv}$.

From Table 2, `DuplexPrinter` is compatible with `HTMLConv`, because the former contains at least one subscription (s_1) that intersects the latter's advertisement (a_3).

3.1.3 Atomic Process

As before, a service W fully satisfies an invocation R, iff R_{adv} precedes W_{subs}, and W_{adv} precedes R_{subs}.

For the services in Table 2, service `SimplePrinter` satisfies invocation `Request1` since `Request1` precedes `SimplePrinter` ($a_5 \bowtie s_0$), and `SimplePrinter` precedes `Request1` ($a_0 \bowtie s_5$).

3.1.4 Composite Process

An invocation R is fulfilled by a composite process represented by a DAG D consisting of a sequence of services $\{W1, W2, \ldots, Wx\} (x >= 2)$ iff

- R_{adv} precedes $W1_{subs}$;
- For every subscription $s \in Wi_{subs} (1 \le i \le x)$, there is an advertisement $a \in \{R_{adv}, W1_{adv}, \ldots, Wi-1_{adv}\}$ where $a \bowtie s$; and
- For every subscription $s \in R_{subs}$, there is an advertisement $a \in (R_{adv}, W1_{adv}, \ldots, Wx_{adv})$ where $a \bowtie s$.

It can be shown that following the above rules will result in the same DAG results as in Figure 3.

3.2 Process Search Algorithm

As described earlier, a service agent registers services in the system by issuing appropriate advertisement and subscription messages into a pub/sub broker (network). This section outlines how the relationships maintained by the pub/sub system among subscriptions and advertisements is exploited to discover a composition of services that satisfy an invocation's input and output requirements.

Before a search for processes fulfilling invocation R, it is assumed that every service W is associated with a service agent, and that the agent has issued the corresponding W_{adv} and W_{subs} messages, and as part of the initialization of the search, the search request agent has issued R_{adv} and R_{subs}.

Recall that by issuing the above messages, the pub/sub system routes the advertisements and subscriptions in such a way the intersection relationships among them form a directed (possibly) cyclic graph G. The objective of the process search algorithm is to discover DAGs within G that satisfy the atomic or composite process properties defined in Section 3.1.

At a high level, the search algorithm works by injecting a publication into the system that is successively delivered to compatible service agents. At each step, a DAG is built by appending a compatible service to the DAG until a DAG is found that originates and terminates at the request agent.

- The request agent issues a publication p, where every predicate constraint in advertisement R_{adv} occurs as an attribute-value pair in p. The publication also includes a representation of a DAG D as a payload. This DAG is successively built as the publication propagates, as described below. At this point, the DAG consists of the single node R.
- Publication p is delivered to all service agents with compatible services. An agent for service W creates a new DAG D' by appending W to the DAG in p, constructs a new publication p' that corresponds to its advertisement W_{adv}, includes D' in p', and injects p' into the pub/sub system.

- If a process fulfilling invocation R is found, a publication p from services compatible with R_{subs} is delivered to the request agent, which appends R to the DAG in p. The resulting DAG describes a process that fulfills the requirements in R, and the DAG is then returned to the user that requested the search.

The algorithm described above omits certain details of the service agent functionality. In particular, the process DAG should consider the successor relation among neighboring services, not compatibility relations. The service agent is responsible for aggregating compatibility relations into suitable successor relations as described below.

3.3 Service Agent

A service agent represents a service, and is responsible for registering the service with the system, and participating in the process search algorithm. A service agent for service W consists of the following components.

- *Pub/Sub client*: This component performs pub/sub messaging, including issuing publications, subscriptions, and advertisements, and receiving matching publications.
- *Publication cache*: Publications from services that are compatible with W but do not precede W are stored in a cache until a set of publications is accumulated that together precede W. This is described in more detail below.
- *Successor matching*: This component executes the algorithm in Figure 5, and is responsible for finding services in the publication cache that precede W.

As outlined above, a service agent waits until it receives a set of publications P_i from services W_i that cumulatively precede the local service W. The algorithm to determine this is given in Figure 5.

```
Algorithm Search(pub)
(* Find a composition of services to append W to. *)
1.  $pubCache \leftarrow pubCache \cup pub$
2.  $md \leftarrow \emptyset$
3.  if for each $s \in W_{subs}$ : $s$ matches $pub$
4.     then $dag \leftarrow pub.payload$
5.          $md \leftarrow dag.append(W)$
6.     else if there exists a minimal set $P \subseteq pubCache$
            such that $\forall s \in W_{subs} : \exists p \in P$ that matches $s$
7.            then for $p \in P$
8.                    do $dag \leftarrow p.payload$
9.                       $dag \leftarrow dag.append(W)$
10.                      $md \leftarrow md.merge(dag)$
11. if $md \neq \emptyset$
12.    then $p \leftarrow$ generatePub($W_{adv}$, $pub$)
13.         $p.payload \leftarrow md$
14.         send($p$)
```

Figure 5: Incrementally search for a process

If a single service W' precedes W, W is simply appended to W' in the DAG, representing a *sequence* relationship between W' and W.

On the other hand, when a set of publications P_i are required to precede W, this represents an *and-join* relationship between the services W_i and W. Therefore, there must be a corresponding *split* relationship in the DAG. The `merge` function in the algorithm in Figure 5 creates these *split* points by merging the largest common prefixes of the DAGs in P_i. The `merge` function can be implemented by a modified topological sort algorithm.

Not shown in Figure 5 is that the publications cached by the service agent expire after some preset time. The publications cannot be removed after a match is found for them because they may be used as part of another result for the same search request. For example, suppose that as part of the search, `DuplexPrinter`'s service agent in Figure 2 receives publications from `Request3` and `HTMLConv`, finds a match, publishes the resulting DAG (in this case the first one in Figure 3), and discards the two publications from its cache. Then when it receives a search publication from `PDFConv` it would no longer discover the second DAG result in Figure 3 because it has discarded the publication from `Request3`.

3.4 Discussion

Some additional details about the distributed automatic service composition algorithm deserve some attention.

3.4.1 Deadlocks

It is possible that there is no composition of services that fulfills an invocation request, that is there are no results. Because the search algorithm is distributed, it is difficult to determine when the search has completed.

A process search with no results has two implications: the requester waits indefinitely, and cached publications at service agents are never flushed. A simple way to resolve these problems is to expire the cached publications after some time, and for the request agent to wait for some bounded time for results. The timeout periods must consider the potential complexity of results (processes that are a composition of many services will typically take longer to find), the distribution of service agents (communication among agents imposes an overhead), and the complexity of the interface representations (interfaces represented by parameter type names can probably be matched quicker by the distributed pub/sub broker network than complex XML schema definitions).

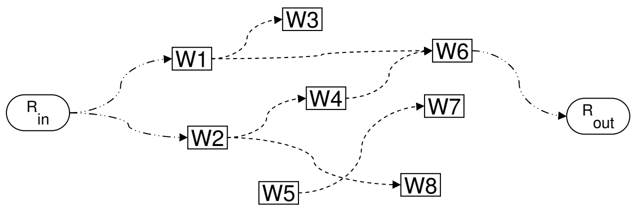


Figure 6: Dead search paths

Note that even in the case where results are found for a process search request, the distributed nature of the search algorithm may still leave unused publications in the publication caches in service agents. This is because not all branches of the search tree reach back to the request agent. For example, consider the compatibility graph in Figure 6 where, for simplicity, it is assumed that every service has exactly one input and output parameter thereby making each compatibility relation a successor relation as well. In this example, a result for the invocation is found as $R_{in} \rightarrow W_2 \rightarrow$

$W_4 \rightarrow W_6 \rightarrow R_{out}$. However, there was also a search path $R_{in} \rightarrow W_1 \rightarrow W_3$ that did not terminate at R_{out}. The publications along these dead paths need to be expired.

3.4.2 *Livelock*

The graph of service compatibility relationships may have cycles, and therefore, the search algorithm may traverse these cycles. For example, for the compatibility graph in Figure 7, an infinite number of DAGs can be found, with zero or more instances of W_4. To avoid this case, a condition is added to the successor matching algorithm in Figure 5 so that the service agent for service W drops publications whose payload contains a DAG that already includes W.

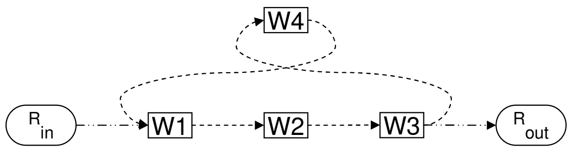


Figure 7: Composition with loop

3.4.3 *Concurrent searches*

There may be multiple requests for a process search, from one or more request agents, occurring simultaneously in the system. To distinguish these searches, the request agent includes a unique request identifier to the publication it sends to trigger the search, and all publications sent by service agents include this identifier. The algorithm in Figure 5 only considers sets of publications with the same identifier when determining predecessor relations. In this way, concurrent searches do not affect one another.

3.4.4 *Reuse search results*

It is possible that processes found as a result of a search may fulfill another search in its entirety or as a part of it. To facilitate these cases, the request agents, can register the resulting processes of a search for invocation $R = \{R_{in}, R_{out}\}$, as another service $W = \{R_{in}, R_{out}\}$. In this case, W is a composite service, and the service agent will append the entire composite process (as opposed to a single service) to the DAG as part of the successor matching algorithm.

3.4.5 *Process constraints*

Process search can be restricted to find processes with constraints such as the maximum depth or maximum number of service compositions. The former can be implemented by using a time-to-live (TTL) field in the search publications that is decremented at every service agent. When the TTL field reaches zero, the publication is discarded, and the search terminates along that path. Another approach is for the service agents to count the number of composed services in the DAG included in the search publication payloads. DAGs with more than the desired services are discarded.

Other constrains include restrictions on the monetary cost of a process (assuming each service imposes some price to invoke it), security restrictions (perhaps processes should not include services from certain combinations of service provides), parallelism (processes with wide *split* and *join* nodes require more computation by process execution engines to collect and aggregate results from services executing in parallel), or search time (a search can be terminated if it is taking longer than the requester is willing to wait for it). Such constraints will be explored in more detail in future extensions to this work.

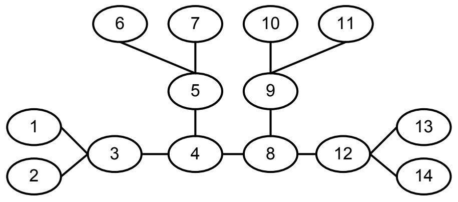


Figure 8: Broker topology

Including such constraints in the distributed search algorithm prunes unnecessary search trees early and avoids delivering invalid results to the requester. This benefits the user and reduces the overhead of the search on the system.

4. EVALUATION

This section experimentally evaluates the automatic service composition algorithm presented in this paper under various scenarios and workloads in order to determine the strengths and weaknesses of the algorithm.

4.1 Setup

The distributed process search algorithm has been implemented over a distributed content-based pub/sub system [8]. The experiments are run in a 14 node cluster of 1.86 GHz machines each with 4 GB of RAM. At most one pub/sub broker is deployed on each machine, and one or more service agents are connected to each broker.

Three deployments, described below, are evaluated.

- *Centralized*: One pub/sub broker is deployed on one machine, with one service agent connected to the broker. This deployment attempts to simulate a centralized process search architecture.
- *Distributed*: One broker is deployed on each of the 14 machines, for a total of 14 brokers forming the overlay shown in Figure 8. As well, 14 service agents are deployed, one per machine, and connected to their local broker. This deployment represents a truly distributed environment and process search algorithm.
- *Hybrid*: One broker is deployed on one machine, and 14 service agents are deployed, one per machine, and

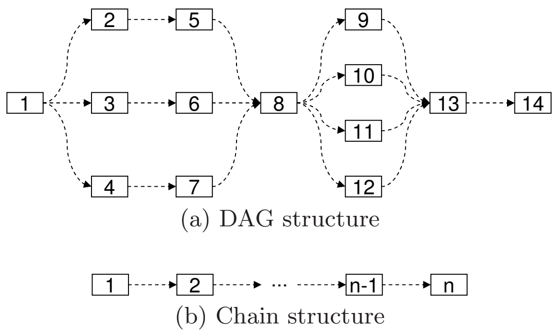
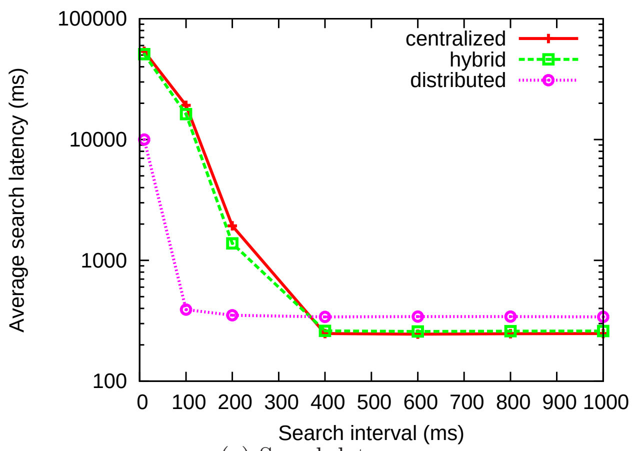
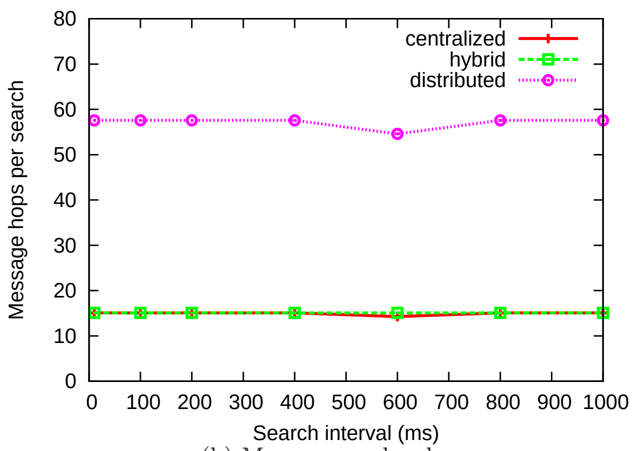


Figure 9: Search result structures



(a) Search latency



(b) Message overhead

Figure 10: Search frequency (Section 4.2.1)

connected to the single broker in the system. This deployment, when compared to the above two, is used to isolate the performance impact on a process search by the pub/sub brokers and service agents.

In all cases above, one request agent is deployed on one machine and connected to the local broker. Unless otherwise specified, 250 services are randomly assigned to the available service agents.

The processes that are results of searches are either a complex DAG shown in Figure 9(a), or a chain of services of variable length illustrated in Figure 9(b). These two process structures are used to study effects such as parallelism, and control factors such as the process length.

The metrics of interest are the search latency and message overhead. The search latency is measured as the duration from when a process search is issued by a request agent to when the agent receives a response. In the results, the latency is typically averaged over a number of search requests. The message overhead counts every hop that every publication traverses in the overlay network, and is normalized to the number of search requests issued to ease comparisons across different experiments. The focus is on publication messages because advertisements and subscriptions are only issued during service registration and are not propagated as part of the search algorithm. The latency measure is probably of more importance to the user, while the message count may be interesting to an administrator of the system.

4.2 Results

The experiments are grouped in terms of the parameter that is varied, and the results are analyzed in detail for each set of experiments.

4.2.1 *Search frequency*

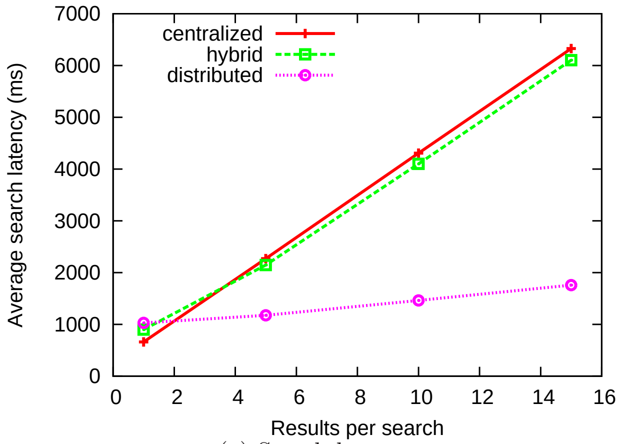
This experiment studies the impact of the number of concurrent process search requests. Each request results in a single result that looks like the process in Figure 9(a). Two hundred requests are issued in total, but the interval between successive requests is varied from 10 ms to 1000 ms.

Figure 10(a) plots the average search latency (on a log scale) for the various request intervals and shows that when the request rate is very low (i.e., large request intervals) the centralized and hybrid deployments outperform the distributed one, and all of the searches take approximately 200 ms to 250 ms. As the request rates are increased, however, the performance of the centralized and hybrid schemes degrade significantly, whereas the distributed deployment remains relatively stable. This is because as the request rates increases, the number of concurrent searches increase as well, and the distributed scheme is able to process the searches in parallel.

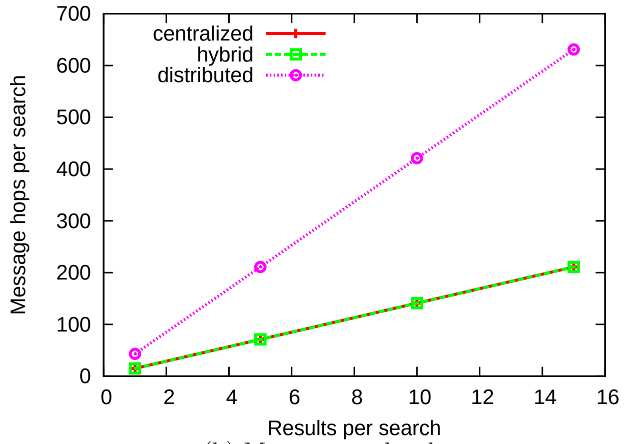
It is interesting to note in Figure 10(a), that the sharp increase in search latency for the centralized and hybrid approaches occurs when the request interval is less than 250 ms, which is roughly the average time it takes to process a single request. In other words, when the request rate exceeds the rate of search evaluation, the system becomes overloaded, queuing delays increase, and search performance degrades drastically. The distributed scheme, however, by parallelizing the search processing, is able to maintain stable search performance even when the request rate exceeds this threshold of 250 ms, but does eventually become overloaded when the request interval is less than 10 ms. Incidentally, note that as there are 14 brokers, an optimal parallelization of the search should take about 250 ms / 14 brokers = 18 ms. Therefore it is expected that a request interval of 10 ms will overwhelm the system.

Figure 10(a) shows that by distributing the service agents (but not the brokers), the hybrid deployment achieves slightly better search latency than the centralized case, but the similarity of their results indicates that the matching done at the pub/sub brokers dominates the processing at the service agents. The similarity between the hybrid and centralized deployments is consistent across all the evaluations.

The message overhead of the three deployments is presented in Figure 10(b). Comparing the three cases, the distributed approach experiences the highest message overhead compared to the centralized and hybrid schemes, which is an intrinsic tradeoff with any distributed algorithm. Notice that there is almost no variance in the message overhead indicating, as expected, that the request interval has no effect on the steps taken by the search algorithm. This supports the argument above that the variations in search latency in

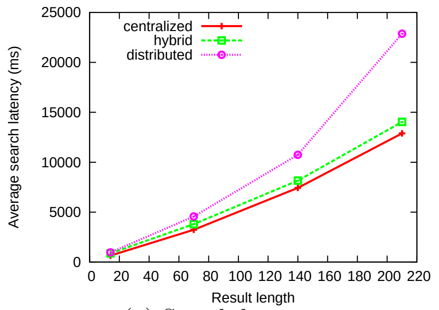


(a) Search latency

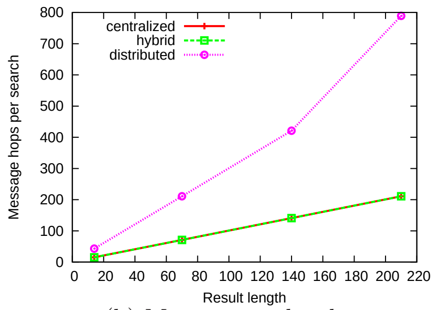


(b) Message overhead

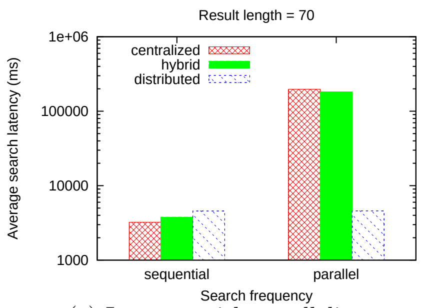
Figure 11: Results per search (Section 4.2.2)



(a) Search latency



(b) Message overhead



(c) Latency with parallelism

Figure 12: Process length (Section 4.2.3)

Figure 10(a) are due to queuing delays (as the system becomes overloaded) and not because of increased matching time or parallelism effects.

4.2.2 *Results per search*

This experiment evaluates the effect of the number of search results per request. A single request is issued each time, but different number of processes are found to match this request. To isolate the effect of the number of results, each result is a simple chain of fourteen services of the structure shown in Figure 9(b).

Figure 11(a) presents the search latency for a varying number of results per search. The latencies for all three approaches are linear with the number of search results. However, the distributed approach benefits from the parallelism opportunities when there are more results per search, and therefore is less sensitive to this parameter.

Notice in Figure 11(b) that although the distributed deployment suffers from the highest message overhead, because these messages are distributed across the available resources, the distributed scheme is still able to provide the lower search latencies in Figure 11(a).

4.2.3 *Process length*

The size of the processes found to satisfy a search has an impact on the performance. In this experiment, requests are issued sequentially (no parallelism), each request returning a process that is a simple chain structure (see Figure 9(b)) but with varying length, so that the number of services composed by the process differs.

The results in Figure 12(a) show that the search latency increases with the result length. Now, the distributed approach performs the worst, and the difference between the distributed and centralized schemes widens with longer paths. The reason for this is because the results in this experiment are chains which afford no opportunity for the distributed approach to parallelize the search; the resulting process is discovered sequentially one service at a time.

To investigate the effects of parallel searches when the results are chains, an additional experiment is run for the 70 process length case, but this time the requests are repeatedly issued with an interval of 1000 ms between each request. Note that this interval is less than the average single request latency of about 3200 ms for the centralized scheme when the process length is 70, so some searches will be processed in parallel. The results in Figure 12(c) compare the sequential and parallel search scenarios, and show that the distributed approach maintains a stable search latency whereas the hybrid and centralized schemes are extremely overloaded (note the log scale in Figure 12(c)).

The results in Figure 12(c) show that when results are long processes, the centralized scheme may outperform the

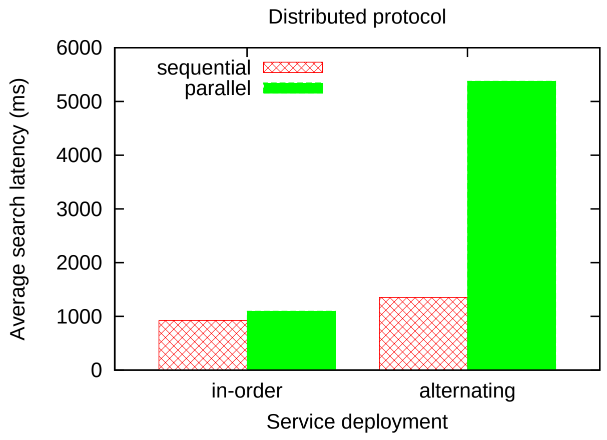


Figure 13: Service deployment (Section 4.2.4)

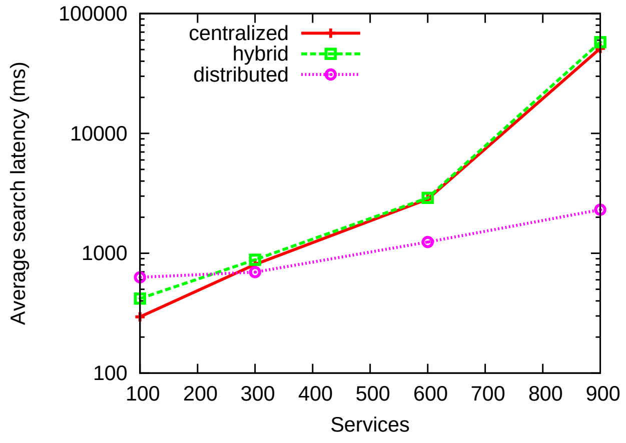


Figure 14: Registered services (Section 4.2.5)

distributed one when there are few concurrent searches, but is not the appropriate deployment choice if many simultaneous searches are expected.

Unlike the experiment from Figure 10(b), the message overhead in this experiment increases with the result length as shown in Figure 12(b). The almost identical trends in Figures 12(a) and 12(b) indicates that the latency increases are primarily due to having to match more messages both in the pub/sub system and the service agents. However, the impact of the service agents is minimal because the hybrid case benefits little from distributing the service agents, and therefore it can be inferred that the pub/sub matching at the brokers is the more significant factor.

4.2.4 *Service deployment*

In the distributed deployment, the location of services in the network may affect the search performance. This experiment repeats the case in Section 4.2.2 where five results are found for a search request, but varies the assignment of services to service agents.

Two extreme service deployment cases are considered: an *in-order* one in which services adjacent in the search result are deployed to service agents at adjacent brokers in the topology in Figure 8; and an *alternating* deployment where consecutive services in the search result chain are assigned to brokers at opposite ends of the network, specifically, to the service agents connected to brokers 1 and 13 in the topology in Figure 8.

As well, two search strategies are used: a *sequential* one where only one request is issued at a time, and a *parallel* one where requests are issued with an interval of 1000 ms, a duration small enough to ensure some searches are processed in parallel.

Figure 13 shows that with sequential requests, the alternating deployment is about 47% worse than the in-order one, whereas the difference is about 390% when requests are issued in parallel. The results show that while both sequential and parallel searches suffer from a poor deployment of services, such as the alternating deployment, the parallel searches are more sensitive to the service distribution because the impact of the extra publications traversing back and forth between the ends of the network is amplified by the number of concurrent searches taking place.

4.2.5 *Registered services*

Each registered service injects a subscription and advertisement into the pub/sub broker network, and imposes additional state at its service agent. Even if a service is not found as part of a search, it may impact the performance of the search because of the matching overhead imposed by its subscriptions and advertisements on the pub/sub brokers. This experiment investigates the effect of "background" services that are registered in the system but are not composed as part of any process.

A single request is issued (no concurrent requests) with a resulting process of the structure in Figure 9(a). Figure 14 presents the average search latency for this request with varying number of "background" services randomly assigned to service agents. The results indicate a large impact from these additional services. The centralized approach clearly does not scale, with more than a 170 fold worse search latency when the number of services is increased from 100 to 900. The distributed deployment scales much better with a sublinear inflation of only about 265% for the 800% increase in the number of services.

4.2.6 *Service similarity*

It is desirable for a process search algorithm to exploit similarities between registered services to prune the search space or optimize the search. Fortunately, there has been much work in pub/sub matching research on exploiting *covering* among subscriptions and advertisements [5].

This experiment repeats the one from Section 4.2.5 with 900 additional "background" services and a single search result of the form in Figure 9(a). However, the similarity of these additional services is controlled. To achieve x% similarity, x% of the services are randomly chosen to be identical to one of five possible services, and the remaining services are mutually different with no common input or output parameters.

Figure 15 shows that in all three deployments, increasing similarity of services results in smaller search latencies. Results show that the message overhead does not vary with similarity, and therefore the latency savings are primarily due to the pub/sub matching algorithm's ability to perform matching faster when there are more covering subscriptions, that is, more services with common interfaces.

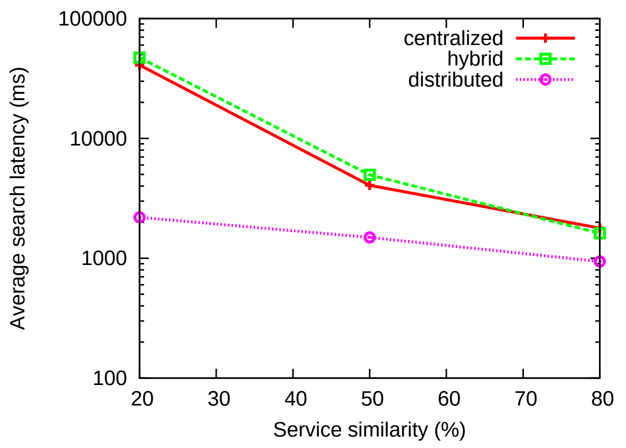


Figure 15: Service similarity (Section 4.2.6)

5. RELATED WORK

This section puts the work in this paper in context with related work in the field of automatic service composition and service computing.

Service modeling: Many models have been developed to facilitate automatic service composition. In one approach Web Services are modeled using DAML-S and DAML+OIL, with a subset of DAML-S defined in first-order logic [17]. Other approaches model services using state transitions as in Roman [4] or Mealy machines in Colombo [3]. An interesting approach models services using Petri nets, and constructs a "service net" with input and output places corresponding to a service's initial and final state, respectively [23]. In this paper, services are modeled using pub/sub messages, mapping the task of constructing service relationships to pub/sub matching, and service composition partly to pub/sub routing. Future work will investigate the possibility of introducing a formal model to support validation and verification of a service composition.

Search techniques: The problem of searching for a composition of services is generally mapped to a planning or a digraph search problem [16]. For example, Bloom filters and A* algorithms can be used to search for chain structured service compositions [18]. An approach to integrate service discovery and automatic service composition has recently been proposed [11]. This integration is also supported by the work in this paper: service discovery can be achieved by performing a process search with the TTL field set to one in order to retrieve only processes containing a single service. Amazon has presented an interesting application scenario for automatic service composition, where two services (Amazon's e-commerce service and an external payment service) export complex interaction protocols and handle structured data in messages, making the composition of services with many interfaces complex and difficult [15].

To the best of our knowledge, existing research only offer centralized architectures for automatic service composition. This paper argues for a potentially more scalable, efficient, and fault-tolerant distributed architecture that avoids single points of failure or bottleneck. A peer-to-peer architecture has been proposed in which the services, a request manager, a matching engine, and an objective engine are separated, but ultimately knowledge of service capabilities and service composition is still centralized [14]. The approach in this paper, on the other hand, enables distributed matching and service composition search by using content-based pub/sub matching and routing.

Search result complexity: The results of an automatic service composition algorithm may range from a simple chain of services to arbitrary DAG structures, with the latter able to return more complex compositions. It has been noted that chain-only results may be "uncertain" if a service's precondition can not uniquely determine a postcondition [21]. An approach to achieve a semantic Web Service composition that can return DAG-like processes as results has been developed [9]. Such processes are also supported by this paper.

Process execution: The processes found by an automatic service composition algorithm will eventually be executed by a centralized or distributed execution engine. The distributed execution of processes and workflows is an active area of research. Self-Serv [2] presents a peer-to-peer architecture that supports the distributed execution of service compositions using distributed routing in the peer-to-peer network. There have also been proposals for using a pub/sub system to achieve distributed process execution [10, 13]. However, none of these systems have considered automatic service composition in a distributed environment. Integrating distributed process search and execution in a unified architecture is an idea that will be explored in future work.

6. CONCLUSIONS

The ability to automatically compose a set of services into a process based on some user-defined criteria is an active area of research that will become more useful as the adoption of service computing grows.

This paper presents, to the best of our knowledge, the first distributed algorithm for automatic service composition. In a novel use of the pub/sub model, service interface specifications are mapped to content-based pub/sub messages in such a way that ordinary pub/sub matching reveals possible service compositions. Based on this mapping, a distributed process search algorithm is developed that exploits the distributed matching capabilities of content-based pub/sub systems. The processing of a search is shared by the pub/sub broker network, which determines possible service relationships, and a set of service agents that collaborate to prune this space and find matching processes or service compositions. The benefits of the distributed architecture include the avoidance of any single point of failure or performance bottleneck, scalability to large numbers of services, and the ability to parallelize searches across distributed resources.

Detailed evaluations of an implementation of the algorithm in a distributed content-based pub/sub system are conducted in a cluster of machines. The distributed algorithm is compared to a simulated centralized one in which all the brokers and service agents are deployed on one physical machine. The results indicate that the distributed algorithm typically scales better and is more stable with respect to the time needed to complete a search. This pattern holds under a variety of conditions, including large numbers of concurrent search requests, increasing results per search, and growing number of registered services. The distributed approach, however, does impose a larger network traffic cost arising from communication overhead among the distributed components. Despite this overhead, the distributed scheme achieves superior performance by parallelizing searches among the resources in the system, including the brokers and service agents. This hypothesis is confirmed by experiments where there is little parallelism possible, such as when there are no concurrent search requests, in which cases the centralized algorithm achieves faster search processing latency.

This paper expands the space of automatic service composition research to distributed architectures and demonstrates the potential benefits of such an approach, but leaves much more work to be explored. Future research avenues include evaluating the proposed algorithm with real-world scenarios, adding support for continuous search, investigating the benefits of integrating process search and process execution architectures, and developing a more formal service interaction model to support features such as process verification.

Acknowledgments

The completion of this research was made possible thanks to Bell Canada's support through its Bell University Laboratories R&D program and thanks to IBM's Center for Advanced Studies. This work builds on the PADRES research project sponsored by CA, Sun Microsystems, the Ontario Centers of Excellence, the Canada Foundation for Innovation, the Ontario Innovation Trust, and the Natural Sciences and Engineering Research Council of Canada.

In addition for part of this work, the first author was supported by the National High-Technology R&D Program, China (Grant No. 2006AA04Z158, 2006AA01A106), and the National Basic Research Program, China (No. 2007-CB310805).

7. REFERENCES

[1] T. Andrews et al. *Business Process Execution Language for Web Services*. 2nd public draft release, Version 1.1, May 2003.

[2] B. Benatallah, Q. Z. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[3] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, pages 613–624, 2005.

[4] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *International Conference on Service Oriented Computing (ICSOC)*, pages 43–58, 2003.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[6] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[7] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[8] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. In *International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI)*, pages 12–30, 2005.

[9] S. Kona, A. Bansal, and G. Gupta. Automatic composition of semantic web services. In *ICWS*, pages 150–158, 2007.

[10] V. Kumar, Z. Cai, B. Cooper, G. Eisenhauer, K. Schwan, M. Mansour, B. Seshasayee, and P. Widener. Implementing diverse messaging models with self-managing properties using IFLOW. In *International Conference on Autonomic Computing (ICAC)*, pages 243–252, June 2006.

[11] U. Küster, B. König-Ries, M. Stern, and M. Klein. Diane: an integrated approach to automated service discovery, matchmaking and composition. In *WWW*, pages 1033–1042, 2007.

[12] G. Li, S. Hou, and H.-A. Jacobsen. Routing of XML and XPath queries in data dissemination networks. In *ICDCS*, 2008.

[13] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*, pages 249–269, 2005.

[14] F. Mandreoli, A. M. Perdichizzi, and W. Penzo. A p2p-based architecture for semantic web service automatic composition. In *International Conference on Database and Expert Systems Applications (DEXA)*, pages 429–433, 2007.

[15] A. Marconi, M. Pistore, P. Poccianti, and P. Traverso. Automated web service composition at work: the amazon/mps case study. In *ICWS*, pages 767–774, 2007.

[16] N. Milanovic and M. Malek. Search strategies for automatic web service composition. *Web Services Research*, 3(2):1–32, 2006.

[17] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW*, pages 77–88, 2002.

[18] S.-C. Oh, B.-W. On, E. J. Larson, and D. Lee. Bf*: Web services discovery and composition as graph search problem. In *International Conference on e-Technology, e-Commerce and e-Service (EEE)*, pages 784–786, 2005.

[19] OSOA. Service component architecture. `http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications`.

[20] J. Rao, D. Dimitrov, P. Hofmann, and N. Sadeh. A mixed initiative approach to semantic web service discovery and composition: Sap's guided procedures framework. In *ICWS*, pages 401–410, 2006.

[21] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. METEOR-S WSDI: A scalable p2p infrastructure of registries for semantic publication and discovery of web services. *Information Technology and Management*, 6(1):17–39, 2005.

[22] Z. Xu and H.-A. Jacobsen. Adaptive location constraint processing. In *SIGMOD*, pages 581–592, 2007.

[23] D. Zhovtobryukh. A petri net-based approach for automated goal-driven web service composition. *Simulation*, 83(1):33–63, 2007.