The Self-Serv Environment for Web Services Composition

The Self-Serv project uses a P2P-based orchestration model to support the composition of multienterprise Web services.

Boualem Benatallah and Quan Z. Sheng
University of New South Wales

Marlon Dumas
Queensland University of Technology

The composition of Web services to handle complex transactions such as finance, billing, and traffic information services is gaining considerable momentum as a way to enable business-to-business (B2B) collaborations.[1] Web services allow organizations to share costs, skills, and resources by joining their applications and systems.[2]

Although current technologies provide the foundation for building composite services, several issues still need to be addressed:

- The development of composite Web services still largely requires time-consuming hand coding, which entails a considerable amount of low-level programming. Because a composite service's components can be heterogeneous, distributed, and autonomous, service composition requires a high-level approach.
- The number of services to be composed can be large and continuously changing. Consequently, approaches that require the service composer to identify, understand, and establish interactions among component services at service-definition time are inappropriate.
- Although the components that contribute to a composite service can be distributed, existing techniques usually employ a central control point. Given the highly distributed nature of services and the large number of network nodes capable of service execution, we believe that novel mechanisms involving scalable and completely decentralized execution of services will become increasingly important.

With the Composing Web Accessible Information and Business Services research project — Self-Serv for short — we are working to develop a middleware infrastructure for the composition of Web services. Our goal is to enhance the potential of Web services by focusing on the dynamic and scalable aspects of their composition.

Self-Serv Web Service Composition

Self-Serv aims to enable the declarative composition of new services from existing ones, the multi-attribute dynamic selection of services within a composition, and peer-to-peer orchestration of composite service executions.

Self-Serv adopts the principle that every service, whether elementary or composite, should provide a programmatic interface based on SOAP and the Web Service Definition Language (see the sidebar "Web Service Composition Background on p. 47). This does not exclude the possibility of integrating legacy applications, such as those written in Corba, into the service's business logic. To integrate such applications, however, first requires the development of appropriate adapters.

The mechanism for composing services in Self-Serv is based on two major concepts: the *composite service* and the *service container*.

Composite Services

In Self-Serv, a composite service is an umbrella structure that brings together other composite and elementary services that collaborate to implement a set of operations. Elementary services provide access to Internet-based applications that do not rely on other Web services to fulfill external requests — for example, a site that provides weather information via SOAP exchanges. In contrast, composite services aggregate multiple component services, such as a travel assistant service that integrates services for booking flights and hotel rooms.

Our system expresses the business logic of a composite service operation as a state chart[3] that encodes a flow of invocations to component service operations. Encoding the flow of operation invocations as state charts has several advantages:

- State charts possess a formal semantics, which is important for unambiguously interpreting and analyzing composite service specifications.
- They have become a well-known and well-supported modeling notation following their integration into the Unified Modeling Language (UML).
- Finally, state charts offer most of the constructs found in contemporary process-modeling languages. Developers can thus adapt service composition mechanisms and orchestration techniques developed in the context of state charts to other process-modeling languages for Web services, such as Business Process Execution Language for Web Services (BPEL4WS) and Web Service Choreography Interface (WSCI).

A state chart is made up of states, which can be either basic or compound, and transitions, which are labeled according to event condition action (ECA) rules.

A basic state corresponds to the execution of a service, whether elementary or composite. Compound states can be either `OR` (containing a single state chart within it) or `AND` (containing several state charts, separated by dashed lines, that are intended to be executed concurrently).

Figure 1 (next page) shows the state charts for four composite services. The state chart of the Complete Travel Planning Service (CTPS) consists of an `AND` state, in which the travel service performs a search for attractions in parallel with the flight and hotel bookings. This `AND` state is followed by an invocation to either a car or a bike rental service, followed by event-planning and payment services. Although we do not discuss it here for space reasons, Self-Serv expresses data-flow dependencies through mappings between data items.[4]

Service Containers

Web services often operate in a highly dynamic environment as providers remove, modify, or relocate their services frequently. In addition, services can form short-term partnerships, disbanding when a partnership is no longer profitable. This form of partnership does not assume any a priori defined relationships between services.

The concept of a *service container* facilitates the composition of a potentially large and changing set of services. A container is a service that aggregates several other substitutable services — those that provide a common capability (the same set of operations) either directly or through an intermediary mapping.

Containers are services in themselves: they are created, advertised, discovered, and invoked just as elementary and composite services are, and they exist independently of the services they aggregate. In particular, in our approach, a basic state in a composite service operation's state chart can invoke a service container operation instead of an elementary or composite service operation. This enables dynamic provider selection: Self-Serv postpones the decision of which specific service handles a given invocation until the moment of invocation. When a requester invokes a service container's operation, the container is responsible for selecting the actual ser-

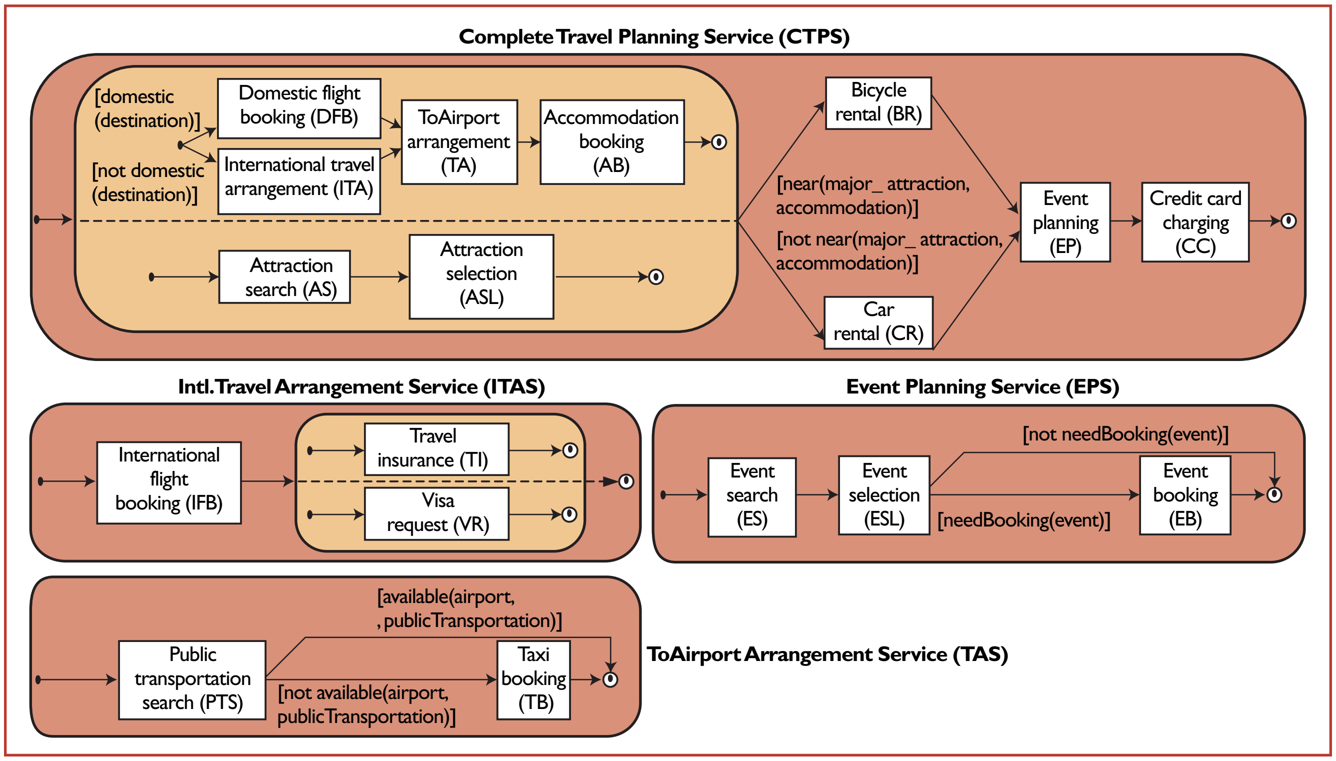
Complete Travel Planning Service (CTPS)
[domestic (destination)]
Domestic flight booking (DFB)
[not domestic (destination)]
International travel arrangement (ITA)
ToAirport arrangement (TA)
Accommodation booking (AB)
Attraction search (AS)
Attraction selection (ASL)
Bicycle rental (BR)
[near(major_ attraction, accommodation)]
[not near(major_ attraction, accommodation)]
Car rental (CR)
Event planning (EP)
Credit card charging (CC)
Intl. Travel Arrangement Service (ITAS)
International flight booking (IFB)
Travel insurance (TI)
Visa request (VR)
Event Planning Service (EPS)
[not needBooking(event)]
Event search (ES)
Event selection (ESL)
[needBooking(event)]
Event booking (EB)
[available(airport, , publicTransportation)]
Public transportation search (PTS)
[not available(airport, publicTransportation)]
Taxi booking (TB)
ToAirport Arrangement Service (TAS)

Figure 1. State charts for four composite services. The Complete Travel Planning Service invokes a number of other services, including the three services shown, to facilitate many aspects of travel preparation.

vice that will execute the operation. To conduct this dynamic selection, a service container uses a membership mode and a scoring service to maintain its list of members and conduct multi-attribute dynamic service selection.

Membership mode. A Web service can be a member of a container in explicit, query, or registration mode. In *explicit mode*, the container's provider sets the collection of services the container can access (its members) at container-definition time. For example, when the container's provider creates a service container `Flight bookings` using the explicit mode, the services it identifies as members of the container will be members for the container's entire life. In *query mode*, the container's provider specifies the collection of members in the form of a query to service registries, such as a UDDI registry. At container-invocation time, the container's provider constructs the collection of services that match the query. In *registration mode*, services need to register with the container to become members. Services can also leave and reenter the container at any time during its lifetime. For a service to register with a container, the service provider must define the mappings between operations defined in the container and those defined in the service, as in the following example.

```
source service Qantas Airway QAS
target container Flight bookings FBS
operation mappings operation
FBS.search_flight() is
QAS.search_ticket();
                        operation
FBS.book_flight() is QAS.book_ticket();
```

This code maps the operation `search_flight` of the container `Flight bookings` to the operation `search_ticket` of the service Qantas Airways, and maps the operation `book_flight` of the container `Flight bookings` to the same service's operation `book_ticket`. A Web service can register with one or more containers, and a container can register with other containers. For example, the Web services Qantas Airways and Cathay Pacific are registered with the container `Flight bookings`, which is, in turn, registered with the container `Intl Travel Arrangements`.

Scoring service. A scoring service lets a container choose a member to execute an operation at container-invocation time by interpreting a selection policy. A selection policy sets a container's service preferences. It consists of a multiattribute utility selection function of the form:

$$U(s) = \sum_{i \in SA} w_i \bullet Score_i(s)$$

where:

- $U(s)$ stands for the value of utility function U for service s,
- $Score_i(s)$ is an attribute scoring function *Score* that, given the value of an attribute i of the service s, returns a positive integer. SA is the set of selection attributes (for example, `price`, `execution_time`, or `reliability`), and
- w_i is the weight w assigned to the attribute i.

The scoring service computes the weighted attribute score and selects the service that yields the highest overall score according to the multiattribute utility function.

The service definition can provide an attribute's value directly, or the scoring service can derive it from information such as the execution logs. The service definition provides the value for the attribute `monetary price`, for example, whereas the attribute `expected execution time` is derived such that, given an operation *op* of a service *s*, the scoring service estimates the time for executing this operation $T(s, op)$ based on past executions.

Each selection attribute has a scoring function. For instance, the scoring function associated with the attribute `execution_time` (*et* for short) is $Score_{et}(s,op) = 1/T(s,op)$ — that is, the higher the execution time, the lower the score. A container might offer several utility functions that correspond to different selection policies. Requesters choose policies based on their preferences, customizing them by providing weights for the selection attributes.

Containers also support functionalities related to change management. Specifically, a service can associate a container with change control policies. This includes operations for monitoring services, subscribing to and notifying services of changes, and rules that specify how to react to change-related events (such as when a member service removes an operation from its interface). Although change management is an important issue, further discussion about this aspect is outside the scope of this article.[5]

Peer-to-Peer Orchestration

Self-Serv takes the view that in order to support scalable execution of composite services over the Internet, services should be *self-orchestrating*: they should be capable of executing composite services without relying on a central scheduler. Accordingly, Self-Serv adopts an orchestration model based on peer-to-peer interactions between software components hosted by the providers participating in the composition. Our preliminary experiments (see the "Performance Evaluation" section on p. 46) have shown that this approach provides greater scalability than those based on a central scheduler because it distributes the runtime message-processing workload across several servers. The two basic elements of this model are the *state coordinator* and *routing table*.

State Coordinators

For each state ST in the state chart of a composite service, Self-Serv generates a state coordinator, which the provider of the service associated with the state ST hosts. At runtime, the coordinator of ST is responsible for

- receiving notifications of completion from other state coordinators and determining from these notifications when to enter state ST;
- invoking the service labeling ST, once all preconditions for entering ST are met, by sending a message to the service and waiting for a reply; and
- notifying the coordinators of the states that might need to be entered next that the service execution is complete.

A coordinator extracts the knowledge it needs to conduct these tasks statically from its routing tables. Self-Serv statically generates these routing tables by analyzing the state chart of the composite service operation.

In addition to generating one coordinator per state for a given composite service description, Self-Serv generates an *initial coordinator*. The initial coordinator is responsible for processing invocations to the composite service, initiating the service, collecting the outputs, and returning them to the application that initiated the invocation.

The orchestration of a composite service execution involves two types of messages: those between the state coordinators and those between the coordinators and the component services. The messages exchanged between the coordinators are called *control-flow notifications*. When a coordinator C1 sends a control-flow notification to a coordinator C2, it signals that the state C1 represents has been executed and that C1 believes the state C2 represents needs to be entered. The notification message contains the composite service execution's input parameters, as well as the current values of the service operation's internal variables. The messages that the coordinator of a state and the service labeling this state exchange are called service invocations or service completions, depending on the action taken. A *service invocation message* contains the name of the service operation being invoked and the values of the input parameters. A *service completion message* contains the values of the return parameters.

When the initial coordinator of a composite service processes a request for executing that service, it sends a control-flow notification to the coordinator of the first state of the composite service's state chart. The coordinator of the first state performs the service invocation labeling its state. Once the execution this invocation induces is completed, the coordinator of the first state sends a control-flow notification to each of the coordinators of the states that might need to be entered next. This process continues until the coordinators of the last states of the composite service state chart send control-flow notifications back to the initial coordinator.

In addition to initiating and completing the execution of a composite service, the initial coordinator is also responsible for detecting and handling failures. For this purpose, it interacts with its peer state coordinators when a timeout occurs, and it processes failure notifications. A state coordinator issues failure notifications when a control-flow notification has not been delivered after several retries.

Routing Tables

The knowledge a coordinator requires is represented in the form of two routing tables: a table of preconditions that must be met before the state is entered, and a table of postprocessing actions indicating which coordinators need to be notified when a state is exited. These tables are defined in a way that requires minimal communication. When a state is exited, the coordinator notifies only those states that might need to be entered next.

The preconditions table of a state ST is a set of rules of the form *E*[*C*] such that:

- *E* is a conjunction of events of the form `ready(ST')`, meaning that the coordinator of ST has received a notification of completion from the coordinator of ST'.
- *C* is a conjunction of conditions appearing in the state chart's transitions.

If an element of the preconditions table is triggered and its condition evaluates to true, the state is entered, resulting in an invocation to the service associated with it. The conjunction *E* of two events e1 and e2 is written e1 · e2. If an occurrence of e1 and an occurrence of e2 are registered in any order, this generates an occurrence of e1 · e2.

In Figure 1, `Preconditions(TA) = {ready(ITA)[true], ready(DFB)[true]}`, which means the state TA is entered when its coordinator receives a control-flow notification from either the coordinator of the state ITA or of DFB. Similarly, `Preconditions(CR) = {ready(AB) · ready(ASL)[not near(major_attraction, accommodation)] }`. The postprocessing table of a state ST is a set of rules of the form [*C*]/*A* such that:

- *C* is a conjunction of conditions appearing in the labels of the state chart's transitions, and
- *A* is a term of the form `notify(ST')`, meaning that the coordinator associated with the state ST needs to send a notification of completion to the coordinator of ST'.

When a service labeling a state completes its execution, the coordinator of that state evaluates the entries appearing in its postprocessing table. For each entry whose condition evaluates to true, the coordinator executes the corresponding notification action.

In Figure 1, `Postprocessing(ASL) = {[true]/notify(CR), [true/notify(BR) }`. The coordinator of ASL cannot evaluate the condition `near(major_attraction , accommodation)` because it involves information that is only known once the accommodation has been selected, which happens in a different thread than the one in which ASL is located.

The algorithms that derive the routing tables of a state by analyzing its incoming and outgoing transitions are described elsewhere.[4]

Self-Serv Architecture and Implementation

Self-Serv adopts a layered architecture to provide support for discovering, creating, composing, and deploying Web services. Figure 2 shows the elements of this architecture, which are grouped in five layers.

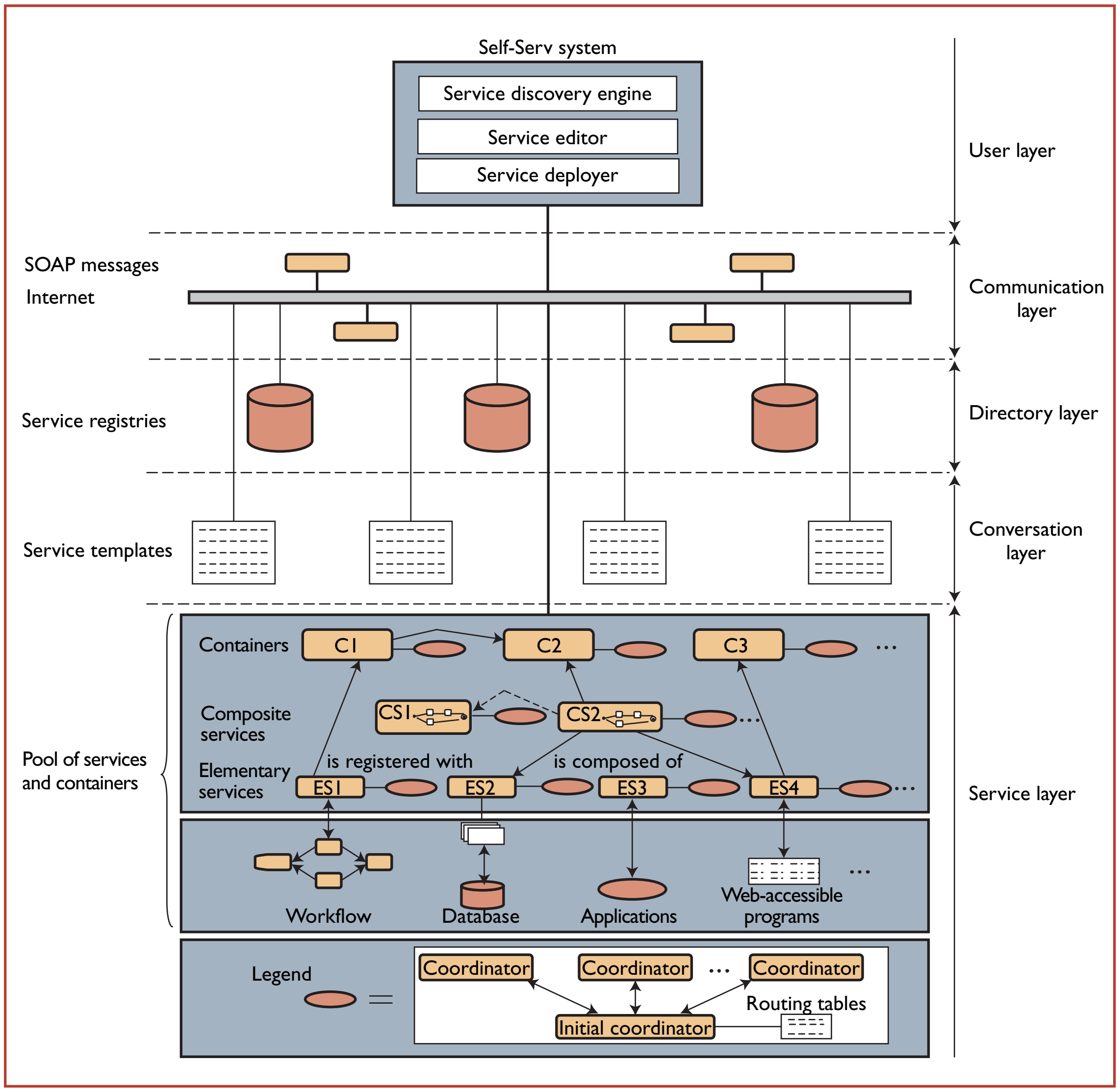

Figure 2. Self-Serv components. The five layers bring together development tools and middleware addressing various aspects of Web services composition.

Service Layer

The service layer consists of a collection of composite services and containers. It features a class `ContainerWrapper` that defines methods for invoking operations provided by containers and collecting the outputs of an invocation. When a user, an application program, or a state coordinator invokes an operation of a container, the `ContainerWrapper` object corresponding to this container invokes the corresponding scoring service. The scoring service takes as input the container's selection policy and the list of members registered with the container, returning the identifier of one of the members.

The service layer also provides two classes, `StateCoordinator` and `InitialCoordinator`, that constitute the runtime environment required to perform peer-to-peer orchestration; service providers must install the classes before participating in a composite service. The `StateCoordinator` class implements methods for receiving, processing, generating, and dispatching control-flow notifications according to a given routing table. In addition to these functionalities, the `InitialCoordinator` class implements methods for invoking an operation of a composite service and collecting the results of the invocation.

Conversation Layer

The conversation layer provides support for standardized interactions among services. For example, it allows business partners to share their external business processes according to a specific B2B standard, such as Electronic Data Interchange

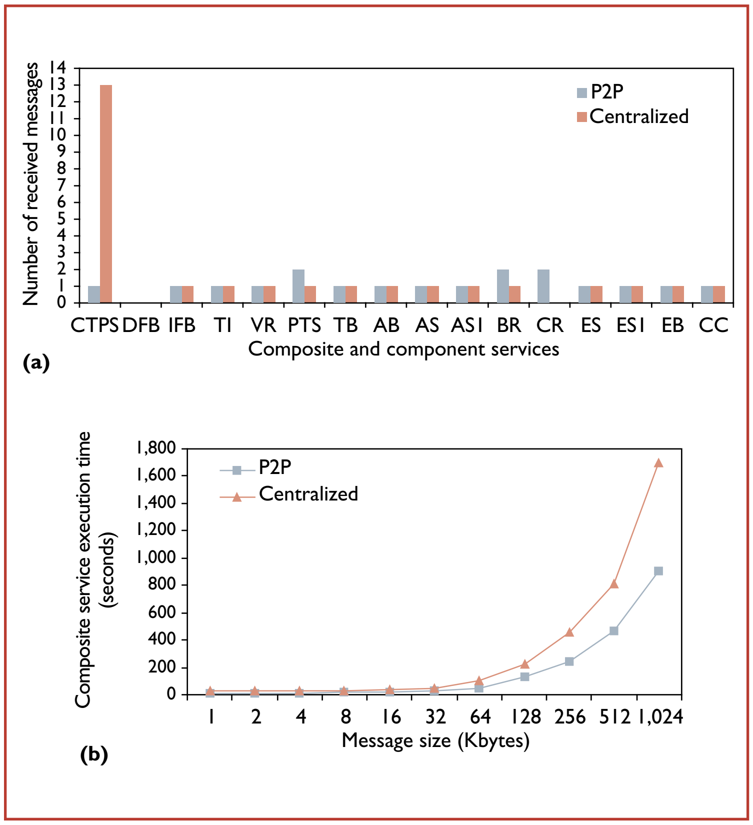

Figure 3. Selected results of the performance evaluation. (a) Workload allocation in the execution of CTPS. (b) Execution time of the composition service CTPS.

(EDI), RosettaNet, or cXML,[6] which defines common formats and semantics for messages (such as request for quote or purchase order) and business process conversations (such as chronology of message exchanges). The conversation layer consists of a set of predefined service templates for various B2B standards.

Directory Layer

The directory layer consists of a set of directories that store metadata about services and containers. Metadata directories contain information that describes the meaning, categories, properties, capabilities, location, and access information of the available services. The user layer of Self-Serv uses the metadata to locate, browse, and query services and containers. Service providers can advertise metadata in service directories such as UDDI or ebXML registries. The current implementation of Self-Serv uses a centralized UDDI registry to store metadata about services and containers.

User Layer

The user layer provides access to the service composition environment through three main components: the *service discovery engine*, the *service builder*, and the *service deployer.*

The service discovery engine facilitates the advertisement and location of services and their operations, which the service builder can then import into service containers or composite services. The discovery engine relies on private UDDI registries.

The service builder allows the developer to create and configure composite services and service containers. It provides an editor for describing registration modes and selection policies for service containers, as well as one for drawing and annotating state chart diagrams for composite service operations. The service builder translates annotated state charts into XML documents for subsequent processing by the service deployer.

The service deployer generates and deploys routing tables for every state in the composite service state charts. It uploads routing tables in XML format to the hosts of the corresponding component services.

Performance Evaluation

We conducted experiments to compare the performance of Self-Serv's P2P orchestration model against a centralized model, as measured by the number of exchanged messages. For each model, the testing executed the CTPS composite service (shown in Figure 1) for every possible combination of truth values for the branching conditions. The results, summarized in Figure 3, showed that the P2P model requires fewer message exchanges for every combination. The average number of physical message exchanges under the P2P model is 15.625, versus 22.125 for the centralized model. Other composite service examples produced similar results.

We also measured the workload allocation of the participating hosts — that is, the host of CTPS and those of its components — by counting the number of messages each node received. The centralized model did not distribute messages evenly as the machine hosting the central scheduler received many messages (13 messages in the example), while the other machines received only one message each. In contrast, the P2P orchestration distributed the workload gracefully among the machines, such that the host of CTPS received only one message. We conducted similar experiments for several composite services and varying message sizes. The P2P orchestration outperformed the centralized orchestration in most cases, regardless of message size.

During the experiments, we also found that having multiple state coordinators orchestrate a

Web Service Composition Background

At present, the technological infrastructure for Web services is structured around three major standards: SOAP, Web Service Definition Language (WSDL), and Universal Description, Discovery, and Integration (UDDI).[1] Other proposed standards, such as the Business Process Execution Language for Web Services (BPEL4WS), Web Service Choreography Interface (WSCI), WS-Coordination, and WS-Transaction (http://dev2dev.bea.com/techtrack/standards.jsp), layer functionality related to composition and transactions on top of the three basic standards.

Standards-Based

SOAP (www.soapclient.com) breaks messages into two parts: *header* and *body*. The header includes information such as intended purpose (service invocation or invocation results, for example), and the body contains an XML representation of a service invocation request or response. WSDL (www.w3.org/TR/wsdl) supports the definition of entry points and message types provided by Web services. It also features the concept of implementation binding as a means of mapping abstract operations to concrete implementations accessible through protocols such as SOAP. UDDI (www.uddi.org) is a registry framework that allows businesses to advertise their services to prospective consumers, who can then select services based on multiple attributes. However, UDDI does not directly support the selection process itself.

BPEL4WS and WSCI describe the control and data flow of composite Web services through process-based Web service composition, which uses process definitions to specify possible interactions and operation invocations between Web services. Orchestration engines such as BPWS4J (www.alphaworks.ibm.com/tech/bpws4j) and Collaxa (www.collaxa.com) assume that knowledge about the control and data-flow dependencies of a composite service is concentrated in a single node that acts as a central scheduler. This approach leads to many unnecessary round-trip messages between the components of a composite service and the central scheduler, increasing the orchestration overhead and creating a bottleneck. Other drawbacks to the centralized execution model include scalability and availability issues. A peer-to-peer orchestration approach like Self-Serv's avoids such problems.

Component-Based

Component-based middleware like CrossWorlds and IBM's SanFrancisco typically relies on distributed object frameworks such as Corba, DCOM, and Enterprise JavaBeans,[2] which are suitable for building robust and secure distributed intraenterprise applications. However, these frameworks require a dedicated software infrastructure whose introduction and maintenance is usually expensive and time-consuming. Among other drawbacks, this hinders the composition of services in these frameworks because not every participant involved in a collaboration will possess the required infrastructure—especially in an interorganizational environment. In contrast, Web services leverage established Internet standards such as HTTP and XML, whose underlying infrastructure is already in place for other purposes. Fortunately, an application developed using component-based middleware can be exported as a Web service, then be composed with other Web services using Web service composition technology. For example, an application developed in Java can be wrapped into a Web service by generating WSDL descriptions from a Java class.

Other projects, including Collaboration Management Infrastructure (CMI),[3] CrossFlow, and eFlow, have explored the idea of declaratively specifying composite services. However, they do not consider the issue of multiattribute dynamic selection of services or peer-to-peer orchestration. Another project related to Self-Serv is DAML-S,[4] which aims to define an ontology for service description, but does not tackle the deployment and execution of composite services.

References

1. F. Curbera et al., "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, Mar. 2002, pp. 86-93.
2. E. Cobb, "The Evolution of Distributed Component Architectures," *Proc. 9th Int'l Conf. Cooperative Information Systems (CoopIS)*, Springer Verlag, New York, pp. 7-21.
3. H. Schuster et al., "Modeling and Composing Service-Based and Reference Process-Based Multi-Enterprise Processes," *Proc. Int'l Conf. Advanced Information Systems Eng. (CAiSE)*, Springer Verlag, New York, 2000, pp. 247-263.
4. A. Ankolenkar et al., "DAML-S: Web Service Description for the Semantic Web," *Proc. 1st Int'l Semantic Web Conf. (ISWC)*, Springer Verlag, New York, 2002, pp. 348-363.

composite service execution created little overhead. Indeed, coordinators are lightweight components that do not require heavy computations because they only need to manipulate small data structures (the routing tables).

Future Directions

Self-Serv leverages emerging Web services standards and an established modeling notation (state charts) to provide high-level support for defining composite Web services involving a variable number of participants. Self-Serv enacts the resulting composite services in a P2P way within a dynamic environment. In addition, the system allows for monitoring and tracing the execution of composite services.[7]

Ongoing research around Self-Serv aims to integrate transaction support for composite Web services, which is hindered by the fact that the components of a composite service can be heterogeneous and autonomous. We plan to extend the descriptions of services by explicitly capturing the transactional semantics of Web service operations, along the lines of WS-Coordination and WS-Transaction (see the sidebar on p. 47).

In order to handle the case when one or more component services fails or is unavailable, we are also considering extending the composition model to integrate transactional semantics for a group of states in a state chart.

Another planned extension to Self-Serv is a visual environment for testing and debugging composite services. This environment will include a module for conducting test deployments, an automated test generator, and a console for interactively displaying the active states of a composite service execution, pausing an execution, and inspecting execution variables and messages.

Acknowledgments

This research has been partly supported by Australian Research Council (ARC) Discovery grant number DP0211207. We would like to thank Eileen Oi-Yan Mak, Nathan Wong, and Alex Yue-Fai Tang for implementing part of this work. We also thank the anonymous reviewers of *IEEE Internet Computing* for the valuable feedback.

References

1. *Distributed and Parallel Databases: An Int'l J.*, special issue on Web services, B. Benatallah and F. Casati, eds, vol. 12, nos. 2-3, Sept. 2002.
2. *VLDB J.*, special issue on e-services, F. Casati, D. Georgakopoulos, and M. Shan, eds., vol. 24, no. 1, Mar. 2001.
3. D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. on Software Eng. and Methodology*, vol. 5, no. 4, Oct. 1996, pp. 293-333.
4. B. Benatallah et al., "Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services," *Proc. Int'l Conf. Data Eng. (ICDE)*, IEEE Press, 2002, pp. 297-308.
5. L. Zeng, B. Benatallah, and A. Ngu, "On-Demand Business-to-Business Integration," *Proc. Int'l Conf. Cooperative Information Systems (CoopIS)*, Springer-Verlag, New York, 2001, pp. 403-417.
6. "Infrastructure for Advanced E-Services," *IEEE Bull. of Technical Committee on Data Eng.*, G. Weikum, ed., vol. 24, no. 1, Mar. 2001.
7. M.C. Fauvet, M. Dumas, and B. Benatallah, "Collecting and Querying Distributed Traces of Composite Service Executions," *Proc. 10th Int'l Conf. Cooperative Information Systems (CoopIS)*, Springer-Verlag, New York, 2002, pp. 373-390.

Boualem Benatallah is a senior lecturer at the University of New South Wales in Sydney, Australia. He received MSc and PhD degrees in computer science from the University of Grenoble, France. Bentallah's latest work focuses on Web services, Web databases, and workflows. He has several publications and funded projects in these areas. He is a member of the IEEE and the ACM. Contact him at boualem@cse.unsw.edu.au.

Quan Z. Sheng is a PhD candidate at the University of New South Wales. He received the BE degree from Beijing University of Aeronautics and Astronautics. Sheng's research interests include Web service composition and mobile Web services. He is a student member of the IEEE Computer Society. Contact him at qsheng@cse.unsw.edu.au.

Marlon Dumas is a lecturer at the Centre for IT Innovation of the Queensland University of Technology in Brisbane, Australia. He obtained a PhD in computer science from the University of Grenoble, France. His research interests are in the areas of Web services, workflow, and e-commerce technologies, with a particular emphasis on Web service composition. Dumas is a member of the IEEE Computer Society. Contact him at m.dumas@qut.edu.au.

NEW for 2003!

IEEE Security & Privacy

Ensure that your networks operate safely and provide critical services even in the face of attacks. Develop lasting security solutions with this new peer-reviewed publication. Top security professionals in the field share information you can rely on:

Wireless Security • Securing the Enterprise
Designing for Security • Infrastructure Security
Privacy Issues • Legal Issues • Cybercrime
Digital Rights Management
Intellectual Property Protection and Piracy
The Security Profession • Education

Don't run the risk!
Be secure.
Order your charter subscription today.

Security & Privacy
Building Confidence in a Networked World
New Challenges, New Thinking
privacy
SECURITY
IEEE

IEEE Computer Society   IEEE

http://computer.org/security