A Research Agenda for Agent-Based Service-Oriented Architectures

Michael N. Huhns

Department of Computer Science and Engineering
University of South Carolina, Columbia, SC 29208, USA
huhns@sc.edu
http://www.cse.sc.edu/~ huhns

Abstract. Web services, especially as fundamental components of service-oriented architectures, are receiving a lot of attention. Their great promise, however, has not yet been realized, and a possible explantion is that significant research and engineering problems remain. We describe the problems, indicate likely directions and approaches for their solution, present an agenda for the deployment of such solutions, and explain the benefits of the resultant deployment. It is our strong expectation that Web services will eventually have an agent basis, which would be needed to address the problems.

1 Introduction

The latest paradigm for structuring large-scale applications is a *service-oriented architecture* (SOA), which involves the linking of small functional services to achieve some larger goal. As the central concept in service-oriented architectures, Web services provide a standardized network-centric approach to making the functionality available in an encapsulated form.

It is worth considering the major benefits of using standardized services. Clearly anything that can be done with services can be done without. So the following are some reasons for using services, especially in standardized form.

- Services provide higher-level abstractions for organizing applications in large-scale, open environments. Even if these were not associated with standards, they would be helpful as we implemented and configured software applications in a manner that improved our productivity and improved the quality of the applications that we developed.
- Moreover, these abstractions are standardized. Standards enable the interoperation of software produced by different programmers. Standards thus improve our productivity for the service use cases described above.
- Standards make it possible to develop general-purpose tools to manage the entire system lifecycle, including design, development, debugging, monitoring, and so on. This proves to be a major practical advantage, because without significant tool support, it would be nearly impossible to create and field robust systems in a feasible manner. Such tools ensure that the components

M. Klusch, M. Rovatsos, and T. Payne (Eds.): CIA 2006, LNCS 4149, pp. 8–22, 2006.

developed are indeed interoperable, because tool vendors can validate their tools and thus shift part of the burden of validation from the application programmer.
– The standards feed other standards. For example the above basic standards enable further standards, e.g., dealing with processes and transactions.

To realize the above advantages, SOAs impose the following requirements:

Loose coupling. No tight transactional properties should generally apply among the components. In general, it would not be appropriate to specify the consistency of data across the information resources that are parts of the various components. However, it would be reasonable to think of the high-level contractual relationships through which the interactions among the components are specified.

Implementation neutrality. The interface is what matters. We cannot depend on the details of the implementations of the interacting components. In particular, the approach cannot be specific to a set of programming languages.

Flexible configurability. The system is configured late and flexibly. In other words, the different components are bound to each other late in the process and the configuration can change dynamically.

Long lifetime. to be useful to external applications, components must have a long lifetime. Morever, since we are dealing with computations among autonomous heterogeneous parties in dynamic environments, we must always be able to handle exceptions. This means that the components must exist long enough to be able to detect any relevant exceptions, to take corrective action, and to respond to the corrective actions taken by others. Components must exist long enough to be discovered, to be relied upon, and to engender trust in their behavior.

Granularity. The participants in an SOA should be understood at a coarse granularity. That is, instead of modeling actions and interactions at a detailed level, it would be better to capture the essential high-level qualities that are (or should be) visible for the purposes of business contracts among the participants. Coarse granularity reduces dependencies among the participants and reduces communications to a few messages of greater significance.

Teams. Instead of framing computations centrally, it would be better to think in terms of how computations are realized by autonomous parties. In other words, instead of a participant commanding its partners, computation in open systems is more a matter of business partners working as a team. That is, instead of an individual, a team of cooperating participants is a better modeling unit. A team-oriented view is a consequence of taking a peer-to-peer architecture seriously.

Web services, viewed as encapsulated and well defined pieces of software functionality accessible to remote applications via a network, are expected to be a fundamental aspect of many future software applications. Many claims have been made about the benefits of Web services for enterprise information systems and

next-generation network-based applications, but the widespread availability and adoption of Web services have not yet occurred. The development, deployment, and proliferation of other new computing technologies can be seen as having occurred in stages as developers and users become familiar with the features of the technology and learn how to exploit them. The development of Web services is likely to progress according to the following four stages.

Stage 1. The first stage in the development of Web services, which is the stage that we are in currently, is that a few specific Web services will be available, mostly on intranets. There will be little or no semantics describing them. Because of this, their discovery will occur manually, their invocation will be hardcoded, and their composition with other Web services will be either nonexistent or done manually. There will be no fees for their use. The resultant applications that make use of the Web services will be brittle and static, but large ones can be crafted relatively quickly. There will be some examples of unexpected uses and utilities.

Stage 2. The second stage in the development of Web services will be characterized by many services being available across the Internet. There will be sufficient semantics, via the use of standardized keywords for narrow domains, to enable semi-automatic discovery. Compostion of services will still be arranged manually. There might be some fees for use, but they will be negotiated off-line by humans. The resultant applications might be dynamic via the substitution of Web services that are explicitly mirrored or via the use of alternative equivalents, most likely where the service functionality is common and straightforward. The desirability of this form of dynamism, and its concomitant robustness, might serve as a major motivation for the further proliferation of Web services and for improved semantics to enable dynamic discovery and a limited form of composition. The scope of the dynamic composition would be one-to-one replacement for a malfunctioning component service.

Stage 3. In the third stage, many Web services will be available, each with a rich semantic description of its functionality. The semantics will enable Web services to be discovered and invoked dynamically and on-demand.

Stage 4. During the fourth stage, some of the many available Web services will be *active*, instead of passive, and will have many of the capabilities that characterize software agents. By being active, they will be able to bid for their use in applications, requiring them to be able to negotiate over Quality-of-Service (QoS) and non-functional semantics. This will include negotiation over fees. In some applications, a candidate Web service could be tried and tested for appropriate functionality and QoS. Comparable services could not only substitute for each other, but also be used redundantly for improved robustness. Moreover, services would self-organize, possible on-demand, into service teams to provide aggregate functionality.

The above four stages of development are driven by limitations of current Web services. These can best be described and understood in the context of the well known "Web service triangle," as shown in Figure 1. In a subsequent section, we consider each of the triangle's vertices and edges and point out the desired enhancements, and thus research, needed for advancement to the next stages.

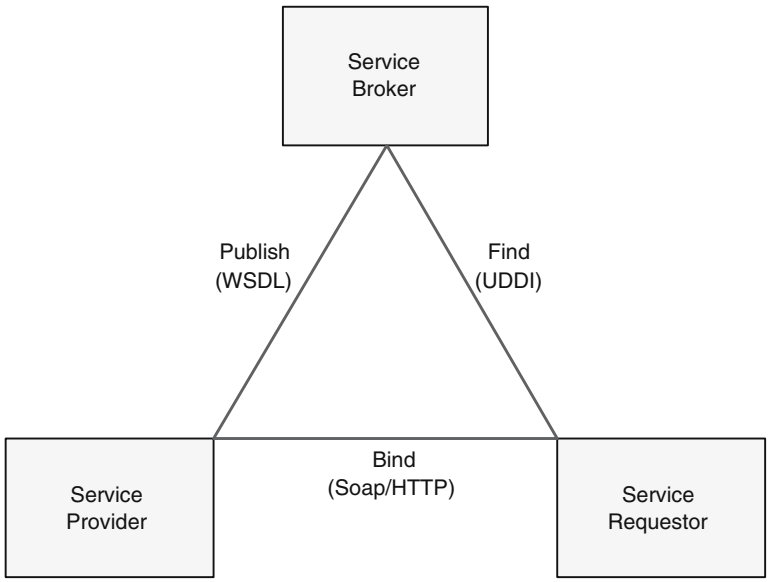

Fig. 1. The familiar Web service triangle. Its limitations can be revealed by considering each of its vertices and edges.

2 An Example of Current SOA Success

To put the above stages in perspective, let's consider a currently successful example of a service-oriented architecture, that of Amazon.com, which serves to explain the great interest in this approach to application development.

Originally, Amazon.com consisted of a large monolithic application that implemented all of the functionality that was made available through its Web site, including the front-end display, back-end database, and business logic [7]. But the monolithic reached a point where it could not be easily scaled to handle the volume of transactions that had to be processed. So, Amazon reengineered it into what became a service-oriented architecture.

Service orientation meant encapsulating the data with the business ligic that operates on it, with the only access through a well defined interface. The services did not share data and did not allow any direct access to the underlying database. The result is hundreds of services and a smaller number of servers that aggregate information from the services. The servers render the Web pages

for the Amazon.com application, as well as serve the customer service application, the seller interface, the external interface to Amazon's Web services, and Amazon-hosted third-party applications. When a user visits the Amazon site, over 100 services are typically invoked in constructing the user's personalized Web page.

The end result is that Amazon can build complex applications quickly out of primitive services, and then scale them appropriately. Moreover, third parties can use the services to build their own applications, primarily for e-commerce, but some for applications unforeseen by Amazon. For example, the Web site "The Amazing Baconizer" invokes Amazon's recommendation seervice for entertainment purposes to list the connections between two items. The connections are done by looking at "people who bought item A also bought item B." Here is a sample result for the connection between the book "Surely You're Joking, Mr. Feynman!" and the DVD "Real Genius":

"Surely You're Joking, Mr. Feynman!" $\Longrightarrow$ "Real Genius" (11 hops):

"Surely You're Joking, Mr. Feynman!" – R. Feynman
$\Longrightarrow$ "Genius: The Life and Science of Richard Feynman," – J. Gleick
$\Longrightarrow$ ...
$\Longrightarrow$ "Weird Science"(DVD) – John Hughes
$\Longrightarrow$ "Top Secret!" (DVD) – Val Kilmer
$\Longrightarrow$ "Real Genius" (DVD) – Val Kilmer

Another interesting third-party service can be accessed from a camera phone. When shopping, a user can take a photo of the bar code for a product, send it to the service, and receive via amazon's services reviews, information on comparable products, and the price.

3 Needed Research for Each Aspect of Web Services

Figure 1 shows the generic architecture for Web services. Although this is a simple picture, it radically alters many of the problems that must be solved in order for the architecture to become viable on a large scale.

- To publish effectively, we must be able to specify services with precision and with greater structure. This is because the service would eventually be invoked by parties that are not from the same administrative space as the provider of the service and differences in assumptions about the semantics of the service could be devastating.
- From the perspective of the registry, it must be able to certify the given providers so that it can endorse the providers to the users of the registry.
- Requestors of services should be able to find a registry that they can trust. This opens up challenges dealing with considerations of trust, reputation, incentives for registries and, most importantly, for the registry to understand the needs of a requestor.

– Once a service has been selected, the requestor and the provider must develop a finer-grained sharing of representations. They must be able to participate in conversations to conduct long-lived, flexible transactions. Related questions are those of how a service level agreement (SLA) can be established and monitored. Success or failure with SLAs feeds into how a service is published and found, and how the reputation of a provider is developed and maintained.

Most of the needed enhancements are related to the scaling of Web services, not only to larger applications but also to more complex environments [1]. That is, there will be a multiplicity of requestors, providers, and registries, whereas the current conception of Web services focuses on just one of each. There will also be more complex interactions than just a simple remote-procedure call to a service: the interactions will be characterized as peer-to-peer, rather than client-server [3].

When there are multiple requestors, then a service provider might be able to share the results of a computation among the requestors, and the requestors might be able to negotiate a "group rate." Of course, this requires the requestors to form a cohesive group and to have a negotiation ability.

Multiple equivalent providers present both a problem and an opportunity. The problem is that a requestor must have and apply a means to choose among them. This would likely require an ability to negotiate over both functional and non-functional attributes (qualities) of the services. The opportunity is that alternatives can yield increased robustness (described more fully below).

Multiple service registries present problems for providers in deciding where to advertise their services and for requestors in deciding where to search for services [4,6]. Also, each registry might employ different semantics and organizations of domain concepts.

Other limitations represented by the current simple model for Web services are

– A Web service knows only about itself—not about its users, clients, or customers
– Web services are not designed to use and reconcile ontologies used by each other or by their clients
– Web services are passive until invoked; they cannot provide alerts or updates when new information becomes available
– Web services do not cooperate with each other or self-organize, although they can be composed by external systems.

Another fundamental problem arises when Web services are composed. Consider the simple example in Figure 2 of one Web service that provides stock quotes in dollars and a second that converts dollars into another currency. Current Web services must be invoked sequentially by a central controller.

A significantly better model is shown in Figure 3, where an interaction protocol under development, WSDL-P, would provide for a continuation by passing a description of an overall workflow through each Web service participating in the workflow. In this way, the composed services would interact directly, rather than through a central intermediary.

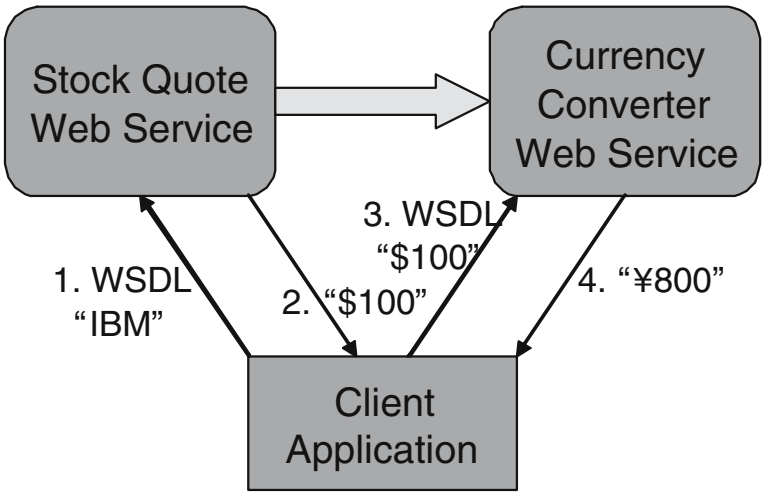

Fig. 2. Composed Web services, which are described traditionally by WSDL

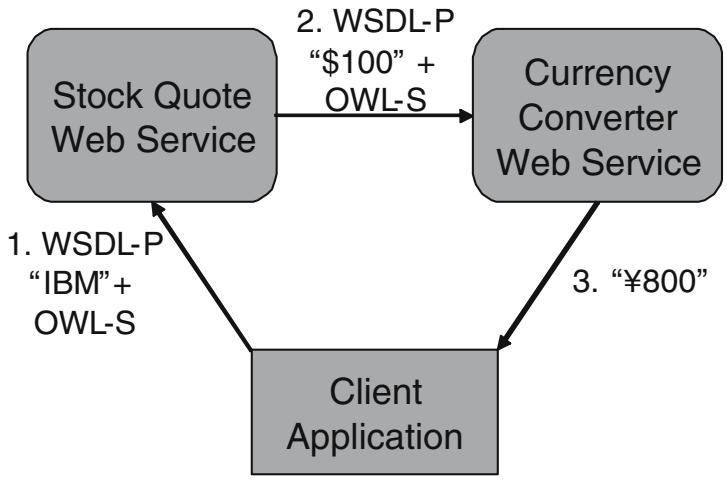

Fig. 3. WSDL-P: Next-Generation Composition. An OWL-S (or BPEL4WS) description of the workflow is communicated through the services

Solutions to the above described problems are under investigation by many research teams. The keys to the next-generation Web are *cooperative services, systemic trust, and understanding based on semantics, coupled with a declarative agent-based infrastructure*. These concepts are elaborated in the next sections, beginning with the notion of commitments as the governing principle for the complex interactions inherent in the later stages of SOA developments.

4 Commitments

For services to apply naturally in open environments, they should be modeled as being autonomous. Autonomy is a natural characteristic of agents, and it is also a characteristic of many envisioned Internet-based services. Among agents, autonomy generally refers to social autonomy, where an agent is aware of its colleagues and is sociable, but nevertheless exercises its independence in certain circumstances. Autonomy is in natural tension with coordination or with the higher-level notion of a commitment. To be coordinated with other agents or to keep its commitments, an agent must relinquish some of its autonomy. However, an agent that is sociable and responsible can still be autonomous. It would attempt to coordinate with others where appropriate and to keep its commitments as much as possible, but it would exercise its autonomy in entering into those commitments in the first place.

The first step to structuring and formalizing interactions among service providers and requestors is to introduce the notion of *directed obligations*, which are obligations directed from one party to another. This is certainly a useful step. Dignum and colleagues describe a temporal deontic logic that helps specify obligations and constraints so that a planner can take deadlines into account while generating plans [2]. However, the approach is based on the notion of obligations, and it does not give operational methods for obligations. Once a deadline has passed and a certain rule has been violated, the logic has nothing to say about the effects on the system. Nevertheless, this approach is semantically rich and detailed in the kinds of deadlines and constraints it allows agents to model. For example, the deadline "as soon as possible," can be modeled.

However, for virtual enterprises and business protocols, it is generally the case that the obligation of one party to another is bounded by the scope of their ongoing interaction. In other words, obligations derived from a virtual enterprise may last no longer than the virtual enterprise in question. Further, there is always the element of conflict, which means that the parties to a contract may be in the need for some adjudication. These considerations suggest that there is an organizational structure to the obligations, which bounds the scopes of the obligations.

The notion of *commitments* (for historical reasons, sometimes referred to as *social commitments*) takes care of the above considerations. Commitments are a legal abstraction, which subsume directed obligations. Importantly, commitments (1) are public, and (2) can thus be used as a basis for compliance. Commitments support the following key properties that make them a useful computational abstraction for service-oriented architectures.

Multiagency. Commitments associate one agent or party with another. The party that "owes" the commitment is called the *debtor* and the other party is called the *creditor*. Each commitment is directed from its debtor to its creditor.

The directionality is simply a representational convenience. In practice, commitments would arise in interrelated sets. For example, a typical business

contract would commit one party to pay another party and the second party to deliver goods to the first party.

Scope. Commitments arise within a well-defined scope. This scope functions as the *social context* of the commitment. In other words, the scope is itself modeled as a multiagent system within which the debtor and creditor of the given commitment interact. For example, the parties to a business contract can be understood as forming and acting in a multiagent system in which they create their respective commitments and act on them. The multiagent system may have a short or a long lifetime depending on the requirements of the application. Conceivably, the multiagent system for a one-off interaction would be dissolved immediately, whereas some multiagent systems may even last longer than the specific agents that belong to them.

Manipulability. Commitments can be acted upon and modified. In particular, commitments may be revoked. If we were to prevent modifying or revoking commitments, we would end up ruling out some of the most interesting scenarios where commitments can be applied. For example, irrevocability would be too limiting for the kinds of open applications where service-oriented architectures make sense. Irrevocability would prevent considering errors and exceptions that may occur outside of the administrative domain of the given business partner. For instance, it may simply be impossible for a vendor to deliver the promised goods on time if the vendor's factory burns down or there are difficulties with shipping. However, we must be careful that commitments are not revoked arbitrarily, which would make them worthless. When restrictions (sensitive to a given context) are imposed on the manipulation of commitments, they can support the coherence of computations.

Services, although collaborative, retain their autonomy. They can exercise their local policies for most decisions and can be considered as being constrained only by their commitments.

4.1 A Formalization of Commitments

We write commitments using a predicate C. A commitment has the form

$$C(x, y, p, G)$$

where x is its debtor, y its creditor, p the condition the debtor will bring about, and G a multiagent system, which serves as the organizational context for the given commitment. A commitment has a simple form, e.g., $C(b, s, \text{pay}(b, s, \$10), D)$, where a buyer b commits to pay \$10 to a seller s a seller within the context of a particular business deal D between b and s.

4.2 Operations on Commitments

It helps to treat commitments as an abstract data type. This data type associates a debtor, a creditor, a condition, and a context. The following are then natural for commitments.

- `create(x, c)` establishes the commitment c in the system. This can only be performed by c's debtor x. For example, x promises to pay \$10 to y.
- `cancel(x, c)` cancels the commitment c. This can be performed only by c's debtor x, for example, x reneges on its promise to pay \$10. Generally, making another commitment compensates cancellation.
- `release(y, c)` releases c's debtor x from commitment c. This only can be performed by the creditor y or a higher authority. For example, x decides to waive receiving the \$10, or the government steps in to say that the agreement is null and void.
- `assign(y, z, c)` replaces y with z as c's creditor. For example, x is now committed to pay \$10 to y's friend z.
- `delegate(x, z, c)` replaces x with z as the debtor for c. For example, now x's friend is committed to pay \$10 to y.
- `discharge(x, c)` means that c's debtor x fulfills the commitment. For example, x actually pays \$10 to y or the assigned creditor.

Create and discharge are obvious; delegate and assign add some flexibility to commitments and are also obvious. Cancel and release remove a commitment from being in effect. Cancel is essential to reflect the autonomy of an agent; just because it made a commitment does not mean that the commitment is irrevocable. However, if commitments could be wantonly canceled, there would be no point in having them, so cancellations of commitments must be suitably constrained. Release helps capture various subtleties of relationships among business partners. A partner may decide not to insist that another party discharge its commitments. Alternatively, the organizational context within which the parties interact may find that a commitment should be eliminated. For example, ordinarily a buyer is expected to pay for goods and a pharmacist is expected to ship medicines that are paid for. However, if the goods arrive damaged then the buyer is released from paying for them (but must return them instead); if the medicine prescription turns out to be invalid, the pharmacist is released from the commitment to ship the medications.

5 Robust Services Via Agent-Based Redundancy

A major driver behind an agent basis for Web services is the demand for robustness. All approaches to robustness rely on some form of redundancy, and Web services are a natural source of redundancy for software applications.

Software problems are typically characterized in terms of bugs and errors, which may be either transient or omnipresent. The general approaches for dealing with them are: (1) prediction and estimation, (2) prevention, (3) discovery, (4) repair, and (5) tolerance or exploitation. Bug estimation uses statistical techniques to predict how many flaws might be in a system and how severe their effects might be. Bug prevention is dependent on good software engineering techniques and processes. Good development and run-time tools can aid in bug discovery, whereas repair and tolerance depend on redundancy.

Indeed, redundancy is the basis for most forms of robustness. It can be provided by replication of hardware, software, or information, e.g., by repetition of communication messages. Redundant code cannot be added arbitrarily to a software system, just as steel cannot be added arbitrarily to a bridge. A bridge is made stronger by adding beams that are not identical to ones already there, but that have equivalent functionality. This turns out to be the basis for robustness in service-oriented systems as well: there must be services with equivalent functionality, so that if one fails to perform properly, another can provide what is needed. The challenge is to design service-oriented systems so that they can accommodate the additional services and take advantage of their redundant functionality.

We hypothesize that agents are a convenient level of granularity at which to add redundancy and that the software environment that takes advantage of them is akin to a society of such agents, where there can be multiple agents filling each societal role [8]. Agents by design know how to deal with other agents, so they can accommodate additional or alternative agents naturally.

Fundamentally, the amount of redundancy required is well specified by information theory. If we want a system to provide n functionalities robustly, we must introduce $m \times n$ agents, so that there will be m ways of producing each functionality. Each group of m agents must understand how to detect and correct inconsistencies in each other's behavior, without a fixed leader or centralized controller. If we consider an agent's behavior to be either correct or incorrect (binary), then, based on a notion of Hamming distance for error-correcting codes, $4 \times m$ agents can detect $m - 1$ errors in their behavior and can correct $(m - 1)/2$ errors.

Redundancy must also be balanced with complexity, which is determined by the number and size of the components chosen for building a system. That is, adding more components increases redundancy, but also increases the complexity of the system.

An agent-based system can cope with a growing application domain by increasing the number of agents, each agent's capability, or the computational and infrastructure resources that make the agents more productive. That is, either the agents or their interactions can be enhanced, but to maintain the same redundancy m, they would have to be enhanced by a factor of m.

N-version programming, also called dissimilar software, is a technique for achieving robustness first considered in the 1970s. It consists of N separately developed implementations of the same functionality. Although it has been used to produce several robust systems, it has had limited applicability, because (1) N independent implementations have N times the cost, (2) N implementations based on the same flawed specification might still result in a flawed system, and (3) each change to the specification will have to be made in all N implementations.

Database systems have exploited the idea of transactions: an atomic processing unit that moves a database from one consistent state to another. Consistent transactions are achievable for databases because the types of processing done are regular and limited. Applying this idea to software execution requires that the state of a software system be saved periodically (a checkpoint) so the system can return to that state if an error occurs.

5.1 Architecture and Process

Suppose there are a number of services, each with strengths, weaknesses, and possibly errors. How can the services be combined so that the strengths are exploited and the weaknesses or flaws are compensated or covered?

Three general approaches are evident in Figure 4. First, a preprocessor could choose the best services to perform a task, based on published characteristics of each service. Second, a postprocessor could choose the best result out of several executing services. Third, the services could decide as a group which ones should perform the task.

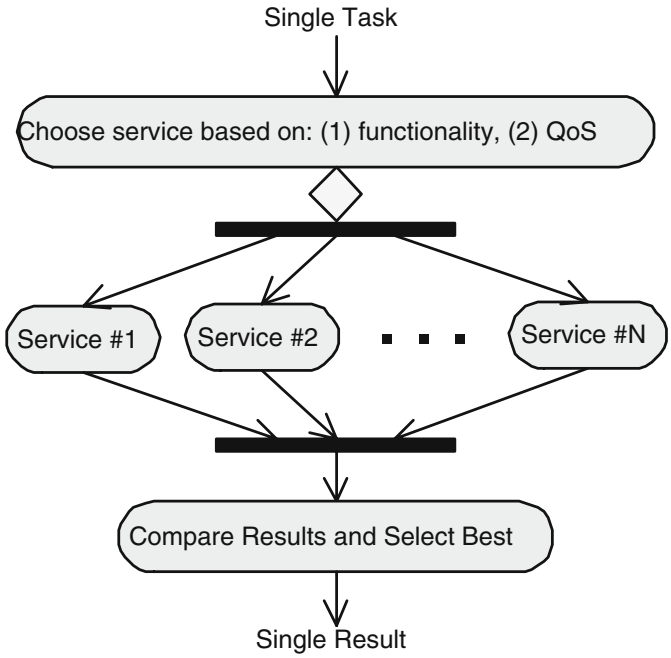

Fig. 4. Improving robustness by combining multiple implementations of a service

The difficulties with the first two approaches are (1) the preprocessor might be flawed, (2) it is difficult to maintain the preprocessor as services are added or changed, and (3) the postprocessor wastes resources, because several services work on the data and their results have to be compared.

The third approach requires distributed decision-making, which is not an ability of conventional Web services. What generic ability could be added to a service to enable it to participate in a distributed decision? The generic capability has the characteristics of an agent, so distributing the centralized functions into the different modules creates a multiagent system. Each agent would have to know its role as well as (1) something about its own service, such as its time and space

complexity, and input and output data structures; (2) the complexity and reliability of other agents; and (3) how to communicate, negotiate, compare results, and manage reputations and trust.

5.2 Experimental Results

Huhns and colleagues collected one set of 25 algorithms for reversing a doubly linked list and another set for sorting a list. Different novice programmers wrote each algorithm. For sorting, no specifications were given to the programmers (beyond that the problem was sorting), so the algorithms all have different data and performance characteristics. For list reversing, the class structure (i.e., method signatures) was specified, so the differences among the algorithms are in performance and correctness.

Each algorithm was converted into an agent, composed of the algorithm written in Java and a wrapper written in Jade. The wrapper knows only about the signature of its algorithm, and nothing about its inner workings.

Our experiments verified that the same wrapper can be used for both the sorting and list-reversing domains. We also verified our hypothesis that more algorithms give better results than any one alone. Further, we investigated both a distributed preprocessor and a centralized postprocessor for combining the agents' functionality, and found that the postprocessor is generally better, but performs worse for large data sets or selected algorithms with long execution times.

The eventual outcome for application development is that service developers will spend more time on functionality development and less on debugging, because different services will likely have errors in different places and can cover for each other.

6 Conclusion and Agenda

Service-oriented computing (SOC) represents an emerging class of approaches with multiagent-like characteristics for developing systems in large-scale open environments. Indeed, SOC presents several challenges that cannot be tackled without agent concepts and techniques. Viewed in this light, SOC offer many ways in which to change the face of computing.

As services become increasingly like agents and their interactions become increasingly dynamic, theyll begin to do more than just manage information in explicitly programmed ways. In particular, services acting in concert can function as computational mechanisms in their own right, thus significantly enhancing our ability to model, design, build, and manage complex software systems. Think of such MASs as providing a new approach for constructing complex applications wherein developers concentrate on high-level abstractions, such as overall behavior and key conceptual structures (the active entities, their objectives, and their interactions), without having to go further into individual agents details or interactions. This vision becomes more compelling as the target environments become more

– populous (a monolithic model is intractable, whereas developers can construct an MAS modularly)
– distributed (pulling information to a central location for monitoring and control is prohibitive, whereas techniques based on interaction among agents and the emergence of desired system-level behaviors are much easier to manage)
– dynamic (an MAS can adapt in real time to changes in the target system and the environment in which it is embedded).

Table 1. Reasons for Complex System Development Based on Multiagent Service-Oriented Systems

Multiagent System Properties	**Benefits for System Development**
Autonomous, objective-oriented behavior; agent-oriented decomposition	Autonomous, active functionality that adapts to the users needs; reuse of whole subsystems and flexible interactions
Dynamic composition and customization	Scalability
Interaction abstractions; statistical or probabilistic protocols	Friction-free software; open systems; interactions among heterogeneous systems; move from sophisticated and learned e-commerce protocols to dynamic selection of protocols
Multiple viewpoints, negotiation, and collaboration	Robustness and reliability
Social abstractions	High-level modeling abstractions

Table 1 shows the ways in which MAS properties can benefit the engineering of complex service-oriented systems. Potential applications and application domains that can also benefit from this approach include meeting scheduling, scientific workflow management, distributed inventory control and supply chains, air and ground traffic control, telecommunications, electric power distribution, water supplies, and weapon systems.

References

1. Mark Burstein, Christoph Bussler, Tim Finin, Michael Huhns, Massimo Paolucci, Amit Sheth, Stuart Williams, and Michael Zaremba, "A Semantic Web Services Architecture," *IEEE Internet Computing*, vol. 9, no. 5, pp. 72–81, September/October 2005.
2. F. Dignum, H. Weigand, and E. Verharen, "Meeting the Deadline: On the Formal Specification of Temporal Deontic Constraints," *Foundations of Intelligent Systems, 9th Intl Symp.*, (ISMIS '96), vol. 1079, Lecture Notes in Computer Science, Springer, 1996, pp. 243–252.
3. A. Eberhart, "Ad-Hoc Invocation of Semantic Web Services," *Proc. IEEE Intl Conf. Web Services*, IEEE CS Press, 2004; www.aifb.uni-karlsruhe.de/WBS/aeb/pubs/icws2004.pdf.

4. K. Sycara et al., "Dynamic Service Matchmaking among Agents in Open Information Environments," *J. ACM SIGMOD Record, Special Issue on Semantic Interoperability in Global Information Systems*, vol. 28, no. 1, 1999, pp. 47–53; http://www-2.cs.cmu.edu/ softagents/papers/ACM99-L.ps.
5. M. Singh and M. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*, John Wiley & Sons, 2005.
6. K. Sivashanmugam, K. Verma, and A. Sheth, "Discovery of Web Services in a Federated Registry Environment," *Proc. IEEE Intl Conf. Web Services*, IEEE CS Press, 2004; http://lsdis.cs.uga.edu/lib/download/MWSDI-ICWS04- final.pdf.
7. Werner Vogels, "Learning from the Amazon Technology Platform," *ACM Queue*, May 2006, pp. 14–22.
8. R.L. Zavala and M.N. Huhns, "On Building Robust Web Service- Based Applications," in *Extending Web Services Technologies: The Use of Multi-Agent Approaches*, L. Cavedon et al., eds., Kluwer Academic, 2004, pp. 293–310.