Chapter 5

Principles of Service-Oriented Computing

The preceding chapters have taken us from a historical perspective of Web services to the basic standards for realizing them, as well as current enterprise computing architectures and programming approaches for implementing Web services. If we were content with using and

fielding simple Web services, the above topics would be more than adequate. However, if we would like to develop and manage systems of real-life complexity, the above concepts are merely a prologue to the underpinnings of the technology that we will need to develop.

The question will come up again and again about the benefits of service-oriented computing and specific approaches for it. There are two major answers to this question. One, service-oriented computing enables new kinds of flexible business applications of open systems that simply would not be possible otherwise. When new techniques improve the reaction times of organizations and people from weeks to seconds, they change the very structure of business. This is not a mere quantitative change, but a major qualitative change. It has ramifications on how business is conducted. Two, service-oriented computing improves the productivity of programming and administering applications in open systems. Such applications are notoriously complex. By offering productivity gains, new techniques make the above kinds of applications practical, thus helping bring them to fruition.

5.1 Use Cases

In order to motivate the full expanse of service-oriented computing that this book presents, it would help to take a closer look at its main use cases—that is, its major application settings. The interesting thing about these use cases is they are not all fancy. This supports the view that service-oriented computing is not a set of futuristic technologies for applications that may or may not matter, but a set of existing and emerging technologies that solve problems that have been with computer science for a while.

To help ground our examples, let us first consider at some length a few aspects of a typical surgery division of a large US hospital. A challenge in such a setting would be to make the payroll, scheduling, and billing systems interoperate. Each of the systems would likely be quite complex and involve its own user interfaces and databases and run on different operating systems.

There are obvious reasons for ensuring that these systems interoperate. For example, scheduling employees and operating rooms for surgery is a complex task; schedules are rarely final and must be updated until the time has passed. This is so the availability of staff and key equipment can be balanced with the unpredictable demand from patients needing surgical procedures with various levels of urgency and advance notice.

Next, the staff must be compensated for their efforts via the payroll system. Just as for scheduling, the mechanisms for payroll are nontrivial, because various kinds of over-time rules would come into consideration for the different categories of labor: nurses, residents, fellows, consulting physicians, senior surgeons, anesthesiologists, radiologists, and so on.

Likewise, the billing system must also incorporate the schedule information. Billing is extremely arcane in most businesses, and especially medicine. The billing system must not only bill customers, but also deal with medical insurance companies and often with government agencies (e.g., agencies for children, the elderly, retired government employees, and veterans). Such agencies typically impose various complex rules for valid billing and penalties for the violation of such rules. For example, different rules apply across the USA as to

how hospitals may bill for the efforts of anesthesiologists during surgeries. In some USA states, a senior anesthesiologist may supervise up to four junior (resident) anesthesiologists. The senior person time-shares his or her effort among four concurrent surgeries, whereas each junior person is dedicated to one of the surgeries. However, if there is an emergency and a fifth surgical procedure must be conducted under the supervision of the same senior anesthesiologist, then that is tolerated, but the billing rate is severely reduced. If such a situation occurs, the billing system must be aware of it. Notice that the billing for a surgery on a patient depends not only on the given surgery, but also on other apparently unrelated events. Conversely, the scheduling system may work with some other decision-support tool to ensure that there are enough anesthesiologists on staff and on call so that the overall billings by the hospital are optimized.

5.1.1 Intraenterprise Interoperation

Intraenterprise interoperation is about achieving the interoperation of applications within an enterprise. This set of problems is also known—mistakenly, in the view of the authors—as enterprise integration. The classical problem here is to make different software components work well together. These components would often have begun their existence as independent, self-contained systems.

Considering only the interoperation aspects of the above problem, we can easily see that several challenges must be overcome. First, there must be connectivity among the applications, nowadays readily ensured through the ubiquity of IP networks. Higher-level protocols such as TCP and HTTP facilitate this further. Second, there must be an ability for the various components to understand each other's communications. The formatting aspect of these are taken care of nowadays through the use of XML, especially with the help of XML Schemas, which capture their syntax. However, challenges remain as to the meanings of the communications. The meaning often is not encoded explicitly, but depends on how the various systems process information. Consequently, one of the greatest challenges to achieving interoperation is reconciling the meanings used in the interacting components. The reconciliation often presupposes that accurate declarative information models exist; in practice, such models often must be reconstructed, because they either would not have been built or were built but not maintained. The information models involve several aspects of abstraction, which are reviewed in Chapter 6.

Because the setting deals with legacy and other systems and application that operate within an enterprise, the parties to the interaction can be more readily authenticated and trusted. Various enterprise policies for authentication and authorization of actions apply and compliance with them can be organizationally enforced, thus simplifying the management of the system.

Service-oriented computing provides the tools to model the information and relate the models, construct processes over the systems, assert and guarantee transactional properties, add in flexible decision-support, and relate the functioning of the component software systems to the organizations that they represent.

5.1.2 Interenterprise Interoperation

Interenterprise interoperation is fast becoming an important application setting for IT. Previously, enterprises interoperated through *ad hoc* means that required substantial human intervention. Or, they used rigid standards such as Electronic Data Interchange (EDI), which led to systems that were difficult to maintain. Recently, there has been growing interest in supply-chain management and intelligent manufacturing, leading up to cross-enterprise processes in general. The idea is that businesses must work together anyway. If they can streamline their interactions through technology, they can improve their responses to information, reduce overhead, and exploit emerging opportunities.

Let us consider our healthcare scenario again. As remarked above, a hospital needs to bill a variety of external entities. These would be insurance companies and government agencies (besides individual patients, whom we can ignore here since we are discussing enterprise interoperation). The traditional approach would be to send printed paper bills, which would be re-keyed in by the receiving party. Naturally enough, such approaches have largely disappeared in favor of online billing.

But online processing exposes a number of challenges of interoperation. The data formats would need to be captured in some reliable manner, so that information formatted by the hospital could be understood by the insurance companies and vice versa. However, considerations of productivity suggest that a standard approach be used, so that the format could be processed via robust commercial tools, rather than through custom software. In recent years, industry has converged on XML as the data format of choice. This is clearly a success. However, it opens up questions of how the content of the data being communicated can be understood and processed.

Service-oriented computing provides the same benefits as for intraenterprise interoperation above. In addition, it provides the ability for the interacting parties to choreograph their behaviors so that each may apply its local policies autonomously and yet achieve effective and coherent cross-enterprise processes.

5.1.3 Application Configuration

Imagine that the hospital purchases an anesthesia information management system (AIMS) to complement its existing systems. An AIMS would enable anesthesiologists to better manage anesthetic procedures on patients in surgery and to better monitor, record, and report key actions, such as the turning on or turning off of various gases and drips. Such information can help establish compliance with government regulations, ensure that certain clinical guidelines are met, and support studies of patient outcomes. It is easy to purchase an AIMS, but not as easy to install and use it (or any such system: this point is not just about AIMS). To introduce such a system requires that the right interface be exposed by the new system and by the existing systems. Since these systems would have been developed on different platforms and may, in fact, run on different operating systems, adherence to standards for interconnections is required for each.

We can assume that low-level connectivity is taken care of. However, it still leaves the

following challenges as in intraenterprise interoperation. One is *messaging* so that the systems can be operationally connected. Another is semantics so that the components understand each other.

However, there is another challenge with introducing a new application. This is to configure and customize its behavior. In the case of an AIMS, it must be populated with a hospital-specific data model or terminology so that it displays the right user interface screens to the hospital staff and logs the right observations. This model is governed by hospital procedures as well as the requirements imposed by the insurance companies and government agencies with whom they interact. If the application is designed with the considerations of service-oriented computing in mind, then it can be quickly configured and introduced into existing business processes.

Service-oriented computing enables the customization of new applications by providing a Web service interface that eliminates messaging problems and by providing a semantic basis to customize the functioning of the application.

5.1.4 Dynamic Selection

Imagine that a hospital wishes to purchase supplies such as catheters. To carry out such purchases efficiently requires that the hospital be able to interoperate with the catheter vendor—a case of interenterprise interoperation. Now suppose that the hospital would like to purchase catheters from whichever vendor offers it the best terms. In other words, the business partner—the other enterprise with which to interoperate—would be chosen on the fly. Such dynamic selection is becoming increasingly common as the possible gains of such flexibility are recognized. If business partners can be selected flexibly, then they can be selected to optimize any kind of quality-of-service criteria, such as performance, availability, reliability, and trustworthiness.

Service-oriented computing enables dynamic selection of business partners based on quality-of-service criteria that each party can customize for itself.

5.1.5 Software Fault Tolerance

Suppose a hospital is carrying out a business transaction with a partner and encounters an error. It would be great if the interaction could be rewired to an alternative business partner dynamically in a manner that is transparent to the overall process. To the extent that the state of the interaction is lost, some means of recovery would be needed to restore a consistent state and resume the computation with new business partners.

Service-oriented computing provides support for dynamic selection of partners as well as abstractions through which the state of a business transaction can be captured and flexibly manipulated; in this way, dynamic selection is exploited to yield application-level fault tolerance.

5.1.6 Grid

Grid computing refers to distributed computing where several resources are made available over a network, and are combined into large applications on demand. Grid computing is a form of *metacomputing* and arose as the successor of previous approaches for large-scale scientific computing. Building complex applications over Grid architectures has been difficult, which has led to an interest in the more modular kinds of interfaces based on services. Accordingly, Grid services have been proposed in analogy with Web services.

Service-oriented computing enables the efficient usage of Grid resources.

5.1.7 Utility Computing

Following up on Grid-like environments, there has been a recent expansion of *utility computing*, where computing resources are modeled as a utility analogous to electric power or telecommunications. The idea is that enterprises would concentrate on their core business and out-source their computing infrastructure to a specialist company. Leading companies such as IBM and HP have made utility computing offerings; the one from IBM is called *autonomic computing*, although that term is starting to be used generically as a technical area for fault-tolerant computing. Utility computing presupposes that diverse computational resources can be brought together on demand and that computations can be realized on physical resources based on demand and service load. In other words, service instances would be created on the fly and automatically bound to dynamically configure applications.

Service-oriented computing facilitates utility computing, especially where redundant services can be used to achieve fault tolerance.

5.1.8 Software Development

Software development remains a challenging intellectual endeavor. Improvements are achieved through the use of superior abstractions. Services offer programming abstractions where different software modules can be developed through cleaner interfaces than before. When the full complement of semantic representations are employed, the resulting modules are not only more easily customizable than otherwise, but the following holds:

Service-oriented computing provides a semantically rich and flexible computational model, for which it is easier to produce software.

5.2 Service-Oriented Architectures

The above use cases provide a challenging set of requirements for any approach to computing. While there are no free lunches in computer science, the requirements can be satisfied more easily through an architecture that matches the essential properties of the above use cases. Let us term such an architecture a service-oriented architecture (SOA).

The emphasis falls on the architecture because many of the key techniques are already well understood in isolation. Practical success would depend on how well these techniques can be placed in a cohesive framework—an architecture—and translated into methodologies and infrastructure so they can be applied in production software development. Recent progress on standards and tools is extremely encouraging in this regard. There can be several SOAs provided they satisfy the key elements of service-oriented computing, which are introduced below.

The current incarnation of Web services emphasizes a single provider offering a single service to a single requester. This is in keeping with a client-server architectural view of the Web.

5.2.1 Elements of Service-Oriented Architectures

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

Loose coupling. No tight transactional properties would 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. The configuration can change dynamically.

Long lifetime. We do not necessarily advocate a long lifetime for our components. However, 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.

Researchers in multiagent systems (MAS) confronted the challenges of open systems early on when they attempted to develop autonomous agents that would solve problems cooperatively, or compete intelligently. Thus, ideas similar to service-oriented architectures were developed in the MAS literature. Although SOAs might not be brand new, they address the fundamental challenges of open systems. Clearly the time is right for such architectures to become more prevalent. What service-oriented computing adds to MAS ideas is the ability to build on conventional information technology and do so in a standardized manner so that tools can facilitate the practical development of large-scale systems.

5.2.2 RPC versus Document Orientation

There are two main views of Web services. Services can be understood in terms of the *RPC-centric view* or the *document-centric view*. The former treats services as offering a set of methods to be invoked remotely, i.e., through remote procedure calls. The latter treats services as exchanging documents with one another. In both views, what is transmitted are XML documents and what is computed with are objects based on or corresponding to the XML documents. However, there is a significant conceptual difference.

The RPC view sees the XML documents as incidental to the overall distributed computation. The documents are merely serializations of the business objects on which the main computation takes place. The document-centric view considers the documents as the main representations and purpose of the distributed computation. Each component reads, produces, stores, and transmits documents. The documents are temporarily materialized into business objects to enable computing, but the documents are the be all and end all of the computation.

The RPC view thus corresponds to a thin veneer of Web services over an existing application. The application determines what functionality the services will support. The document view more naturally considers Web services as a means of implementing business relationships. The documents to be processed (and their relationships) determine the functionality of the services. The business objects, such as there are, on either side of a relationship are local, and should not be exposed to the other side.

For this reason, the document-centric view coheres better with our primary use case of applying services in open environments. The RPC view is more natural for the use case of making independently developed applications interoperate. What happens is that application developers expose their application interface in the form of Web services, which can then be bound to in the usual manner. If the applications are designed for method integration, then the RPC view of services is natural for such interoperation. However, if the applications are

designed—as they should be—to function as independent components, then the document-centric view would be natural even for application interoperation.

5.3 Major Benefits of Service-Oriented Computing

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

- 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 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.

5.4 Composing Services

Although there can be some value in accessing a single service through a semantically well-founded interface, the greater value is clearly derived through enabling a flexible *composition* of services. Composition leads to the creation of new services from old ones and can potentially add much value beyond merely a nicer interface to a single preexisting service. The new services can be thought of as *composite services*.

From a business perspective too, intermediaries that primarily offer access to a single service would have a tough time thriving or even surviving. Airline travel agents are a case in point. Traditional travel agents provide a nice user interface: friendly and with a human touch, but little more. However, airlines do not like to pay commissions for services that merely repackage their offerings. As a result, the airlines compete with the travel agents and reduce their commissions, slowly squeezing them out of business. This is as one would expect where the offerings are conceptually simple, especially for frequent customers. By

contrast, package tour operators, who combine offers from airlines and other vendors, can prosper. In other words, the increased complexity due to subtle compositions is essential for intermediaries to flourish, because it provides an opportunity for offering greater value to customers.

Service composition concepts involve enough intricacy as to attract considerable interest and to demand a careful analysis of the underlying principles. The need for principles is greater as the basic infrastructure for Web services becomes more common. We address these principles herein. Sometimes, the term *composition* is taken to mean a particular approach to achieving composition, for example, by invoking a series of services. In the present usage, however, composition refers to any form of putting services together to achieve some desired functionality.

Composed Web services find application in a number of practical settings. For example, portals aggregate information from a number of sources and possibly offer programmatic facilities for their intended audience. The challenge to making an effective portal is to be able to personalize the information presented to each user. Electronic commerce is another major scenario where users would like to aggregate product bundles to meet their specific needs. Virtual enterprises and supply-chain management reflect generalizations of the consumer-oriented e-commerce scenarios, because they include more subtle constraints among a larger number of participants.

5.4.1 Goals of Composition

Most of the applications touted for Web services are simple and straightforward client-server interactions. For example, an airline's flight-schedule database could interact directly with a PC user's personal information and appointment software to book a flight; or software for a personal database of contacts could automatically query a distant series of phone databases to add missing numbers to its list. This sort of a scenario is not far-fetched at all. Even in the early days of Web service standards, Southwest Airlines and Dollar Rent-A-Car developed a prototype system that used SOAP to link Southwest's Web site to Dollar's reservation system, so that airline customers could reserve a car along with their airline tickets [Metz, 2001].

Although useful, such applications are insufficient to drive the strong development and deployment of Web services. The fruitful, and also the more challenging, applications require services to be combined in ways that yield more powerful and novel uses. For example, the airline's flight schedule might also enable a fuel supplier to anticipate fuel purchases by the airline and to alert its refineries to adjust their production rates. Or, a travel agency Web site could combine the services of Southwest Airlines, Dollar Rent-A-Car, and Sheraton to construct custom travel packages.

Service composition has been studied in the research literature for quite some time, but it is now becoming an important theme in practical Web system development. The basic idea behind service composition is simple. Web sites can be thought of as not only offering content, but also providing services. For example, Yahoo! provides a news service and Amazon provides a book selection service. We typically invoke these services by hand through a

Web browser, but a program could invoke them directly. Service composition on the Web is about taking some existing services and building new customized services out of them. For instance, you might find the latest news headlines and search for books that match those headlines. Another example is where you might take the news from one service, filter it through a service that selects news based on a given user's interests, and pass the selected news items through a transcoding service to create a personalized Web page that a user could review through a handheld device. Or, more conventionally, you could create a travel service that invokes hotel, airline, and car rental services. In other words, you would create a workflow over the existing services.

5.4.2 Challenges for Composition

The main advantage of Web services arises when we can compose them to create new services. Unfortunately, much of the attention on Web services has been focused on the lower-level, infrastructural matters, often down to encoding syntaxes and unnecessarily narrow means of invoking services. For Web services to be composed effectively requires an understanding of deeper concepts. These concepts have been developed in diverse parts of computer science, especially heterogeneous databases, distributed computing, artificial intelligence, and multiagent systems.

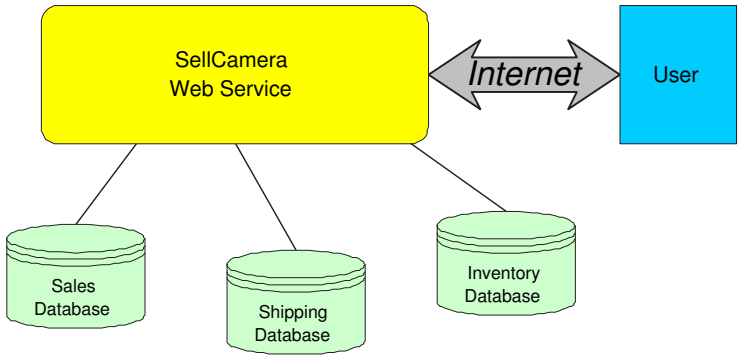

Figure 5.1: An example of a business-to-consumer (B2C) transaction environment, where cameras are sold to customers over the Web

Consider a simple business-to-consumer (B2C) situation, where a company sells digital cameras over the Web, combining an on-line catalog with up-to-date models and prices, a valid credit-card transaction, and a guaranteed delivery. The B2C transaction software, as shown in Figure 5.1, would

- Record a sale in a sales database.
- Debit the credit card.

- Send an order to the shipping department.
- Receive an OK from the shipping department for next-day delivery.
- Update an inventory database.

However, some problems can arise: What if the order is shipped, but the debit fails? What if the debit succeeds, but the order was never entered or shipped? If this were a closed environment, then transaction processing (TP) monitors (such as IBM's CICS, Transarc's Encina, or BEA System's Tuxedo) can ensure that all or none of the steps are completed and that the systems eventually reach a consistent state.

But suppose the user's modem is disconnected right after he clicks on OK. Did the order succeed? Suppose the line went dead before the acknowledgment arrives. Will the user order again? The basic problem is that the TP monitor cannot get the user into a consistent state! The user is part of the software system's environment, which is open because it can accommodate any user. The TP monitor is able to control no more than the part of the environment that is within its scope.

Possible solutions for an open environment include

- The server application could send email about credit problems, or detect duplicate transactions.
- A downloaded Java applet could synchronize with the server after the broken connection was reestablished and recover the transaction; the applet could communicate using HTTP, or directly with server objects via CORBA's Internet Inter-ORB Protocol (IIOP) (where an ORB is CORBA's object request broker) or Remote Method Invocation (RMI).
- If there are too many orders to process synchronously, they could be put in a message queue, managed by a Message Oriented Middleware (MOM) server (which guarantees message delivery or failure notification), and customers would be notified by email when the transaction is complete.

Email is typically used for people to communicate with each other, so in using email, the server is behaving like an intelligent agent. We will have more to say about the emerging agent-like aspects of the Web and its services in later chapters.

Notice that although the above example considers a user dealing with a particular enterprise, the problem arises in more acute form in business-to-business settings. If our little camera store were considered as merely a component in a large supply network, it would have no hope of forcing the other parties to conduct their local transactions in any particular manner or to reliably converge to a state that would be consistent across the system. Deeper models of transactions and of business processes are needed to ensure that the correct behavior is realized in such cases.

The current specifications for Web services do not address transactions or specify a transaction model. The Organization for the Advancement of Structured Information Standards

(OASIS) is developing one, but the view of most implementors is that SOAP will manage transactions—somehow. Without guidance from a standard or an agreed-upon methodology by the major vendors, transactions will be implemented in an *ad hoc* fashion, thus defeating the hopes for interoperability and extensibility.

Some of the other problems for composed services are

- Security will be more difficult, because more participants will be involved and the nature of their interactions and their needs might be unanticipated by the designers of the services.
- There will be incompatibilities in vocabularies, semantics, and pragmatics among the service providers, service brokers, and service requesters.
- As services are composed dynamically, performance problems might arise that were not anticipated.
- Dynamic service composition will make it difficult to guarantee the quality of service (QoS) that applications require.

Two fundamental styles for delivering Web services are emerging, characterized as RPC-style (favored by Sun) and document-style (favored by Microsoft, and supported by Sun). In the latter style, the body of a SOAP message would not have the call-response semantics of most programming languages, but rather would consist of arbitrary XML documents that use WSDL to describe how a service works. In the long term the document-style is likely to prevail, because it is more declarative (rather than procedural), more asynchronous, and more consistent with the document-exchange underpinnings of the Web.

5.5 Spirit of the Approach

Figure 2.1 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.

The keys to the next-generation Web are *cooperative services, systemic trust, and understanding based on semantics, coupled with a declarative agent-based infrastructure*.

The size and dynamism of the Web presents problems, but it fortuitously provides a means for solving its own problems. For example, for a given topic there might be an overload of information, with much of it redundant and some of it inaccurate, but a system can use voting techniques to reduce the information to that which is consistent and agreed upon. For another example, there might be many potential service providers competing for many potential clients, and some of the providers might not be trustworthy, but a system can use a Web-based reputation network to assess credibility. Finally, there might be many different ontologies used by different sites, but a multiplicity of ontologies can be shown to yield a global, dynamically formed, consensus ontology. We address each of these concepts in subsequent chapters.

5.6 Notes

5.7 Exercises

5.1. Consider the basic Web service architecture and its main components introduced above: WSDL, SOAP, and UDDI. List and briefly explain a total of six shortcomings of these three components. Two sentences each would be adequate to convey the essential points, although you could certainly present more in-depth analyses.