Software Agents: The Future of Web Services

Michael N. Huhns

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

Abstract. The World-Wide Web is evolving from an environment for people to obtain information to an environment for computers to accomplish tasks on behalf of people. The resultant Semantic Web will be computer-friendly through the introduction of standardized Web services. This paper describes how Web services will become more agent-like, and how the envisioned capabilities and uses for the Semantic Web will require implementations in the form of multiagent systems. It also describes how the resultant agent-based Web services will yield unprecedented levels of software robustness.

1 Introduction

I recently transfered the title for my daughter's car from my name to hers. This straightforward transaction involved the following flurry of documents and interactions with various agencies: I needed tax receipts from my county and my state, a cancelled check for taxes paid, a driver's license verification from the motor vehicle department, a new license plate for the car from the license plate bureau, a proof of disposal of the old license plate, the old title, and, finally, the new title. The county tax agency has strict rules about issuing duplicate receipts, but these rules are difficult to understand and not fully known by my state motor vehicle department.

What does this have to do with Web services? Well, the organizations participating in my tax-and-title transaction might have implemented their capabilities as on-line Web services. By invoking each other's functionalities, the Web services could have determined the necessary forms, the required fees, and the verification of the identities of myself and my daughter. A single visit to exchange license plates physically would have been sufficient, I would have received a new title on-line, and I could have authorized payment via an automatic debit from my bank account.

Because of the potential illustrated above, Web services are the hottest trend in information technology: it is hard to find a computer magazine today that doesn't feature them. Web services are XML-based, work through firewalls, are lightweight, and are supported by all software companies. They are a key component of Microsoft's .NET initiative, and are deemed essential to the business directions being taken by IBM, Sun, and SAP.

Web services are also central to the envisioned *Semantic Web* [1], which is what the World Wide Web is evolving into. But the Semantic Web is also seen as a friendly environment for software agents, who will add capabilities and functionality to the Web. What will be the relationship between agents and Web services?

1.1 The Semantic Web

The World-Wide Web was designed for humans. It is based on a simple concept: information consists of pages of text and graphics that contain links, and each link leads to another page of information, with all of the pages meant to be viewed by a person. The constructs used to describe and encode a page, the Hypertext Markup Language (html), describe the appearance of the page, but not its contents. Software agents don't care about appearance, but rather the contents.

There are, however, some agents that make use of the Web as it is now. A typical kind of such agent is a *shopbot*, an agent that visits the on-line catalogs of retailers and returns the prices being charged for an item that a user might want to buy. The shopbots operate by a form of "screen-scraping," in which they download catalog pages and search for the name of the item of interest, and then the nearest set of characters that has a dollar-sign, which presumably is the item's price. The shopbots also might submit the same forms that a human might submit and then parse the returned pages that merchants expect are being viewed by humans. The Semantic Web will make the Web more accessible to agents by making use of semantic constructs, such as ontologies represented in DAML, RDF, and XML, so that agents can *understand* what is on a page.

The World-Wide Web was designed for people to get information, such as finding out about books; the Web also supports people getting work done, such as buying a book. In its current form, the Web has the following characteristics:

- HTML describes how things appear
- HTTP is stateless
- Sources are independent and heterogeneous
- Processing is asynchronous and client-server
- There is no support for integrating information
- There is no support for meaning and understanding

The envisioned Web services of the Semantic Web are expected to be

- Robust
- Composable
- Dynamic
- Distributed
- Aware of client's needs so that they can volunteer their services.

1.2 Current Standards for Web Services

There are a number of definitions for Web services. For example, a Web service is said to be

- ...a piece of business logic accessible via the Internet using open standards..." (Microsoft)
- Loosely coupled software components that interact with one another dynamically via standard Internet technologies (Gartner)
- A software application identified by a URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts and supports direct interactions with other software applications using XML-based messages via Internet-based protocols (W3C)

My working definition is: A Web service is functionality that can be engaged over the Web.

Web services are currently based on the triad of functionalities depicted in Figure 1. The architecture for Web services is founded on principles and standards for connection, communication, description, and discovery. For providers and requestors of services to be connected and exchange information, there must be a common language. This is provided by the eXtensible Modeling Language (XML).

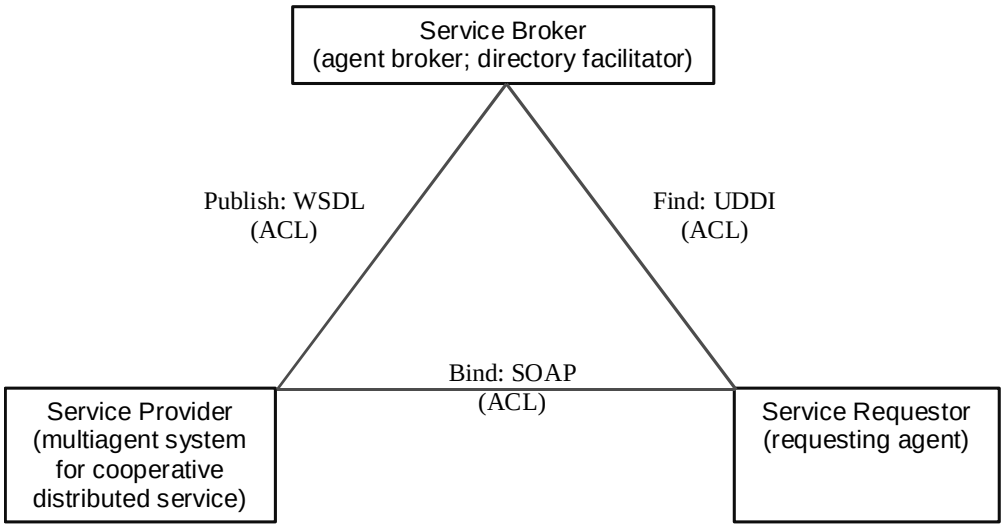

Fig. 1. The general architectural model for Web services. Web services rely on the functionalities of publish, find, and bind. The equivalent agent-based functionalities are shown in parentheses, and all interactions are via an agent-communication language (ACL)

A common protocol is required for systems to communicate with each other, so that they can request services, such as to schedule appointments, order parts, and deliver information. This is provided by the Simple Object Access Protocol (SOAP) [4].

The services must be described in a machine-readable form, where the names of functions, their required parameters, and their results can be specified. This is provided by the Web Services Description Language (WSDL).

Finally, clients—users and businesses—need a way to find the services they need. This is provided by Universal Description, Discovery, and Integration (UDDI), which specifies a registry or "yellow pages" of services.

Besides standards for XML, SOAP, WSDL, and UDDI, there is a need for broad agreement on the semantics of specific domains. This is provided by the Resource Description Framework (RDF) [7, 8], the DARPA Agent Modeling Language (DAML), and, more generally, ontologies [12].

1.3 Directory Services

The purpose of a directory service is for components and participants to be able to locate each other, where the components and participants might be applications, agents, Web service providers, Web service requestors, people, objects, and procedures. There are two general types of directories, determined by how entries are found in the directory: (1) name severs or *white pages*, where entries are found by their name, and (2) *yellow pages*, where entries are found by their characteristics and capabilities.

The implementation of a basic directory is a simple database-like mechanism that allows participants to insert descriptions of the services they offer and query for services offered by other participants. A more advanced directory might be more active than others, in that it might provide not only a search service, but also a brokering or facilitating service. For example, a participant might request a brokerage service to recruit one or more agents who can answer a query. The brokerage service would use knowledge about the requirements and capabilities of registered service providers to determine the appropriate providers to which to forward a query. It would then send the query to those providers, relay their answers back to the original requestor, and learn about the properties of the responses it passes on (e.g., the brokerage service might determine that advertised results from provider X are incomplete, and so seek out a substitute for provider X).

UDDI is itself a Web service that is based on XML and SOAP. It provides both a white-pages and a yellow-pages service, but not a brokering or facilitating service.

The DARPA DAML effort has also specified a syntax and semantics for describing services, known as DAML-S. This service description provides

- Declarative ads for properties and capabilities, used for discovery
- Declarative APIs, used for execution
- Declarative prerequisites and consequences, used for composition and interoperation.

2 Foundations for Web Services

Current Web services have either a database or a programming basis. Both are unsatisfactory. To illustrate the database basis and its shortcomings, consider the following simple business-to-customer Web service example: suppose a business wants a software application to sell cameras over the Web, debit a credit card, and guarantee next-day delivery. The application must

- 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

What if the order is shipped, but the debit fails? What if the debit succeeds, but the order was never entered or shipped? A traditional database approach works only for a closed environment:

- Transaction processing monitors (such as IBMs CICS, Transarcs Encina, BEA Systems Tuxedo) can ensure that all or none of the steps are completed, and that systems eventually reach a consistent state
- But what if the users modem is disconnected right after he clicks on OK? Did the order succeed? What if the line went dead before the acknowledgement arrives? Will the user order again?

The essential problem is that the transaction processing 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. In more modern approaches designed for open environments, a server application could send email about credit problems, or detect duplicate transactions. A downloaded applet could synchronize with the server after a broken connection was restored and then recover the transaction; the applet could communicate using http, or directly with server objects via CORBA/IIOP or RMI.

If there are too many orders to process synchronously, they could be put in a message queue, managed by a Message Oriented Middleware server (which guarantees message delivery or failure notification), and customers would be notified by email when the transaction is complete. In essense, *the server behaves like an agent!*

With a programming basis for Web services, software is partitioned into composable services, which are invoked by an application using, for example, RMI. This is illustrated in Figure 2. In this figure, suppose application A invokes service B, but B is busy and delegates the request to service C. When service C sends a response to A, A *fails* because it expected a response from B.

3 Composing Cooperative Web Services

Imagine that a merchant would like to enable a customer to be able to track the shipping of a sold item. Currently, the best the merchant can do is to point the

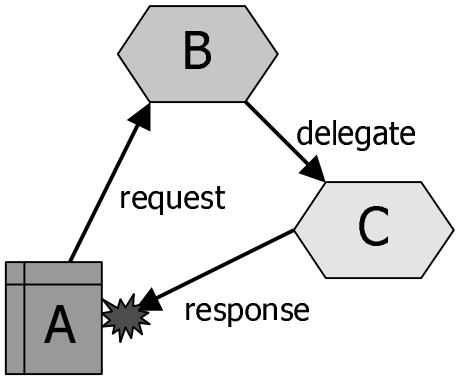

Fig. 2. An illustration of a programming basis for Web services

customer to the shipper's Web site, and the customer can then go there to check on delivery status. If the merchant could compose its own production notification system with the shipper's Web services, the result would be a customized delivery notification service by which the customer—or the customer's agents—could find the status of a purchase in real time.

As Web uses (and thus Web interactions) become more complex, it will be increasingly difficult for one server to provide a total solution and increasingly difficult for one client to integrate solutions from many servers. Web services currently involve a single client accessing a single server, but soon applications will demand federated servers with multiple clients sharing results. Cooperative peer-to-peer solutions will have to be managed, and this is an area where agents have excelled. In doing so, agents can balance cooperation with the interests of their owner.

Composing Web services requires capturing patterns of semantic and pragmatic constraints on how services may participate in different compositions. It also requires tools to help reject unsuitable compositions so that only acceptable systems are built. The following challenges have not yet been met by current implementations and standards for Web services:

Information Semantics The composer and the member services must agree on the semantics of the information that they exchange.

Collaboration To perform even simple protocols reliably, service providers must ensure that the parties to an interaction agree on its current state and where they desire to take it. This requires elements of teamwork through (1) persistence of the computations, (2) ability to manage context, and (3) retrying.

Autonomous Interests Services should be able to participate in automated markets, where various mechanisms are required for effective participation. This requires abilities to (1) set prices, (2) place bids, (3) accept or reject bids, and (4) accommodate risks.

Personalization Effective usage of services often requires customization of the compositions in a manner that is context sensitive, especially with respect to user needs. This requires (1) learning a customer's preferences, (2) mixed-initiative interactions, offering guidance to a customer (best if it is not intrusive) and letting a user interrupt the composed service, and (3) acting on behalf of a user, which is limited to ensure that a user's autonomy is not violated.

Exception Conditions To construct virtual enterprises dynamically in order to provide more appropriate, packaged goods and services to common customers requires the ability to (1) construct teams, (2) enter into multiparty deals, (3) handle authorizations and commitments, and (4) accommodate exceptions.

Service Location Recommendations must be provided to help customers find relevant, high quality, and trustworthy services. This requires a means to (1) obtain evaluations, (2) aggregate evaluations, and (3) find evaluations.

Distributed Decision-Making Decision-making will be distributed across the composed services, which requires intelligent decisions by each service so the composed services can collaborate and compete appropriately. The objective is to achieve the desired composition while accommodating exceptions.

With these concerns being addressd by various research efforts, the Web will evolve from being passive to active, client-server to peer-to-peer to cooperative, services to processes, and semantics to mutual understanding to pragmatics and cognition. The result, as indicated in Figure 3, will be a Semantic Web that enables work to get done and better decisions to be made.

4 Agents Versus Web Services

Typical agent architectures have many of the same features as Web services. Agent architectures provide yellow-page and white-page directories, where agents advertise their distinct functionalities and where other agents search to locate the agents in order to request those functionalities. However, agents extend Web services in several important ways:

- A Web service knows only about itself, but not about its users/clients/customers. Agents are often self-aware at a metalevel, and through learning and model building gain awareness of other agents and their capabilities as interactions among the agents occur. This is important, because without such awareness a Web service would be unable to take advantage of new capabilities in its environment, and could not customize its service to a client, such as by providing improved services to repeat customers.
- Web services, unlike agents, are not designed to use and reconcile ontologies. If the client and provider of the service happen to use different ontologies, then the result of invoking the Web service would be incomprehensible to the client.

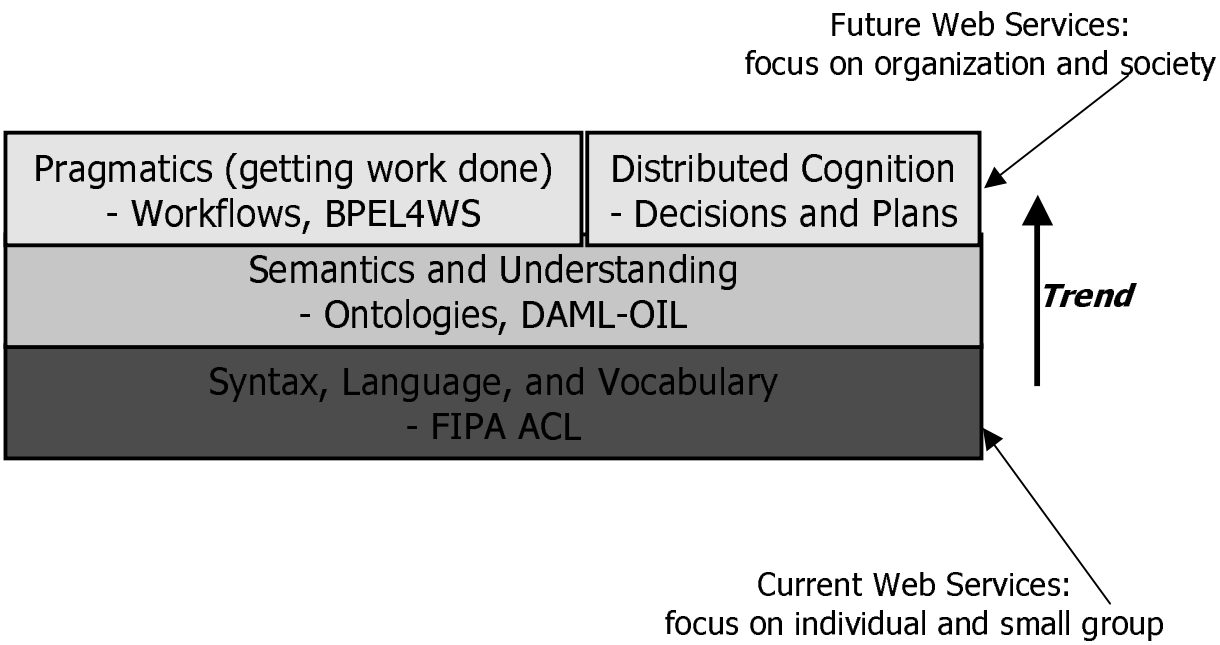

Fig. 3. Beyond the Semantic Web

– Agents are inherently communicative, whereas Web services are passive until invoked. Agents can provide alerts and updates when new information becomes available. Current standards and protocols make no provision for even subscribing to a service to receive periodic updates.

– A Web service, as currently defined and used, is not autonomous. Autonomy is a characteristic of agents, and it is also a characteristic of many envisioned Internet-based applications. 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.

– Agents are cooperative, and by forming teams and coalitions can provide higher-level and more comprehensive services. Current standards for Web services do not provide for composing functionalities.

4.1 Benefits of an Agent-Oriented Approach

Multiagent systems can form the fundamental building blocks for software systems, even if the software systems do not themselves require any agent-like behaviors [19]. When a conventional software system is constructed with agents as its modules, it can exhibit the following characteristics:

- Agent-based modules, because they are active, more closely represent real-world things
- Modules can hold beliefs about the world, especially about themselves and others
- Modules can negotiate with each other, enter into social commitments to collaborate, and can change their mind about their results
- Modules can volunteer to be part of a software system.

The benefits of building software out of agents are [5, 15]

1. Agents enable dynamic composibility, where the components of a system can be unknown until runtime
2. Agents allow interaction abstractions, where interactions can be unknown until runtime
3. Because agents can be added to a system one-at-a-time, software can continue to be customized over its lifetime, even potentially by end-users
4. Because agents can represent multiple viewpoints and can use different decision procedures, they can produce more robust systems. The essence of multiple viewpoints and multiple decision procedures is redundancy, which is the basis for error detection and correction.

4.2 Advanced Composition

Suppose an application needs simply to sort some data items, and suppose there are 5 Web sites that offer sorting services described by their input data types, output date type, time complexity, space complexity, and quality:

1. One is faster
2. One handles more data types
3. One is often busy
4. One returns a stream of results, while another returns a batch
5. One costs less

An application could take one of the following possible approaches:

- Application invokes services randomly until one succeeds
- Application ranks services and invokes them in order until one succeeds
- Application invokes all services and reconciles the results
- Application contracts with one service after requesting bids
- Services self-organize into a team of sorting services and route requests to the best one

The last two require that the services behave like agents. Furthermore, the last two are scalable and robust, because they take advantage of the redundancy that is available.

5 Redundancy and Robustness

Redundancy is the basis for most forms of robustness. It can be provided by replication of hardware, software, and information, and by repetition of communication messages. For years, NASA has made its satellites more robust by duplicating critical subsystems. If a hardware subsystem fails, there is an identical replacement ready to begin operating. The space shuttle has quadruple redundancy, and won't leave the ground without all copies functioning. However, software redundancy has to be provided in a different way. Identical software subsystems will fail in identical ways, so extra copies do not provide any benefit.

Moreover, code cannot be added arbitrarily to a software system, just as steel cannot be added arbitrarily to a bridge. When we make a bridge stronger, we do it 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 software systems as well: there must be software components with equivalent functionality, so that if one fails to perform properly, another can provide what is needed. The challenge is to design the software system so that it can accommodate the additional components 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 [13]. Agents by design know how to deal with other agents, so they can accommodate additional or alternative agents naturally. They are also designed to reconcile different viewpoints.

Fundamentally, the amount of redundancy required is well specified by information and coding theory. Assume each software module in a system can behave either correctly or incorrectly. Then two modules with the same intended functionality are sufficient to detect an error in one of them, and three modules are sufficient to correct the incorrect behavior (by voting, or choosing the best two-out-of-three). This is exactly how parity bits work in code words. Unlike parity bits, and unlike bricks and steel bridge beams, however, the software modules can't be identical, or else they would not be able to correct each other's errors.

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, $4m$ agents can detect $m - 1$ errors in their behavior and can correct $(m - 1)/2$ errors.

Fundamentally, redundancy must 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 might also increase the complexity of the system. This is just another form of the common software engineering problem of choosing the proper size of the modules used to implement a system. Smaller modules are simpler, but their interactions are more complicated because there are more modules.

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

To underscore the importance being given to redundancy and robustness, several initiatives are underway around the world to investigate them. IBM has a major initiative to develop autonomic computing—"a systemic view of computing modeled after the self-regulating autonomic nervous system." Systems that can run themselves incorporate many biological characteristics, such as self-healing (redundancy), adaptability to changing environments (reconfigurability), identity (awareness of their own resources), and immunity (automatic defense against viruses). An autonomic computing system will adhere to self-healing, not by "cellular regrowth," but by making use of redundant elements to act as replenishment parts. By taking advantage of redundant services located around the world, a better range of services can be provided for customers in business transactions.

5.1 N-Version Programming

N-version programming, also called dissimilar software, is a technique for achieving robustness first considered in the 1970's. It consists of N disparate and 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) the resultant system might have N times the maintenance cost (e.g., each change to the specification will have to be made in all N implementations).

5.2 Transaction Checkpointing, Rollback, and Recovery

Database systems have exploited the idea of transactions for maintaining the consistency of their data. A transaction is an atomic unit of processing that moves a database from one consistent state to another. Consistent transactions are achievable for databases because the types of processing done are very regular and limited.

Applying this idea to general software execution requires that the state of a software system be saved periodically (a checkpoint) so that the system can return to that state if an error occurs. The system then returns to that state and processes other transactions or alternative software modules. This is depicted in Figure 4.

There are two ways of returning to a previous state: (1) reloading a saved image of the system before the recently failed computation, or (2) rolling back, i.e., reversing and undoing, each step of the failed computation. Both of the ways suffer from major difficulties:

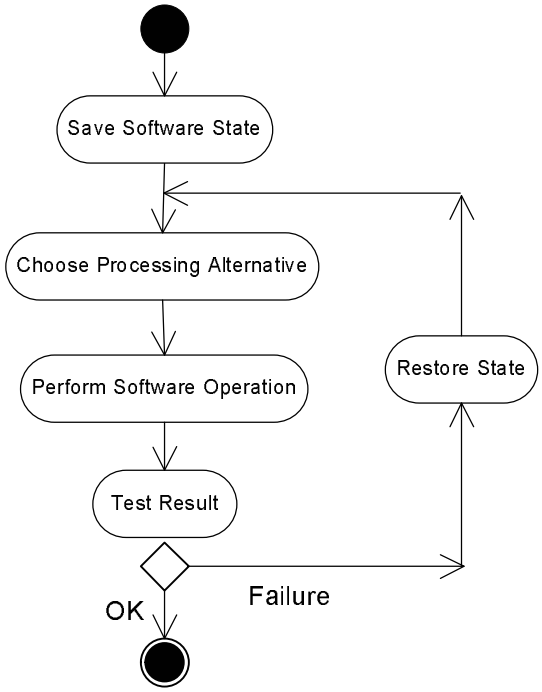

Fig. 4. A transaction approach to correcting for the occurrence of errors in a software system

1. The state of a software system might be very large, necessitating the saving of very large images
2. Many operations cannot be undone, such as those that have side-effects. Examples of these are sending a message, which cannot be un-sent, and spending resources, which cannot be un-spent. Rollback is successful in database systems, because most database operations do not have side-effects.

5.3 Compensation

Because of this, compensation is often a better alternative for software systems. Figure 5 depicts the architecture of a robust software system that relies on compensation of failed operations.

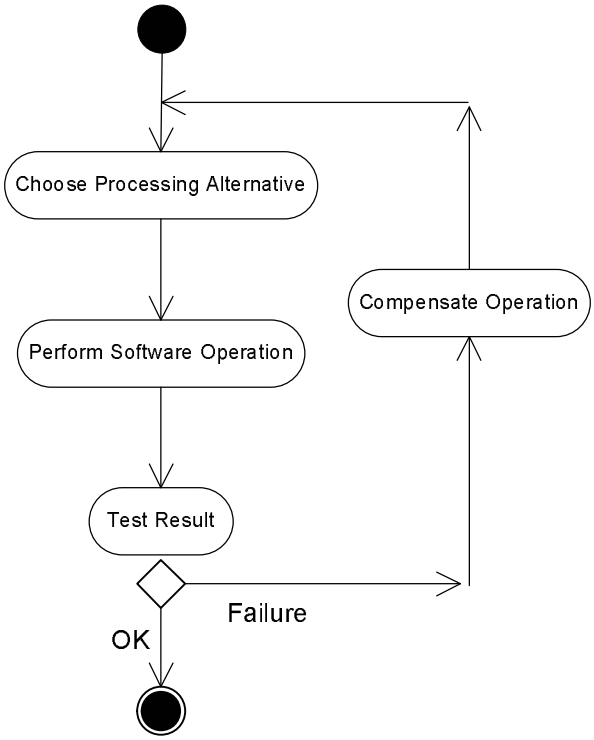

Fig. 5. An architecture for software robustness based on compensating operations

6 Architecture and Process

Suppose there are a number of sorting algorithms available. Each might have strengths, weaknesses, and possibly errors. One might work only for integers,

while another might be slower but be able to sort strings as well as integers. How can the algorithms be combined so that the strengths of each are exploited and the weaknesses or flaws of each are compensated or covered? In solving this in a general way, we hypothesize that the end result is an "agentizing" of each algorithm.

A centralized approach, as shown in Figure 6, would use an omniscient preprocessing algorithm to receive the data to be sorted and would then choose the best algorithm to perform the sorting. Each module's characteristics have to be encoded into the central unit. The central unit can use a simplistic algorithm for determining the best, based on known facts about each of the modules. The difficulties with this approach are (1) the preprocessing algorithm might be flawed and (2) it is difficult to maintain such a preprocessing algorithm as new algorithms are added and existing algorithms become unavailable. Also, only one module at-a-time executes, there is low CPU usage, and results are taken as-is when completed.

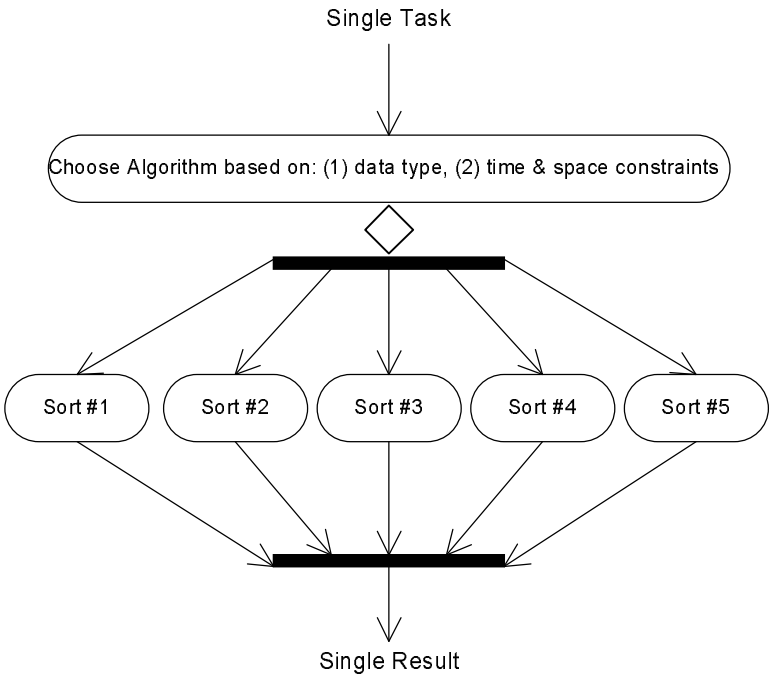

Fig. 6. Centralized architecture for combining N versions of a sorting algorithm into a single, more robust system for sorting, where a preprocessing algorithm chooses which sorting algorithm will execute

An improvement might be a postprocessing algorithm, as shown in Figure 7, that receives the results of all sorting algorithms and chooses the best result to

be the output. Results have to be compared and voted on to determine the best. This approach is also centralized and suffers from a waste of CPU resources, because all algorithms work on the data. However, due to the comparison of outcomes, it is likely to produce better results.

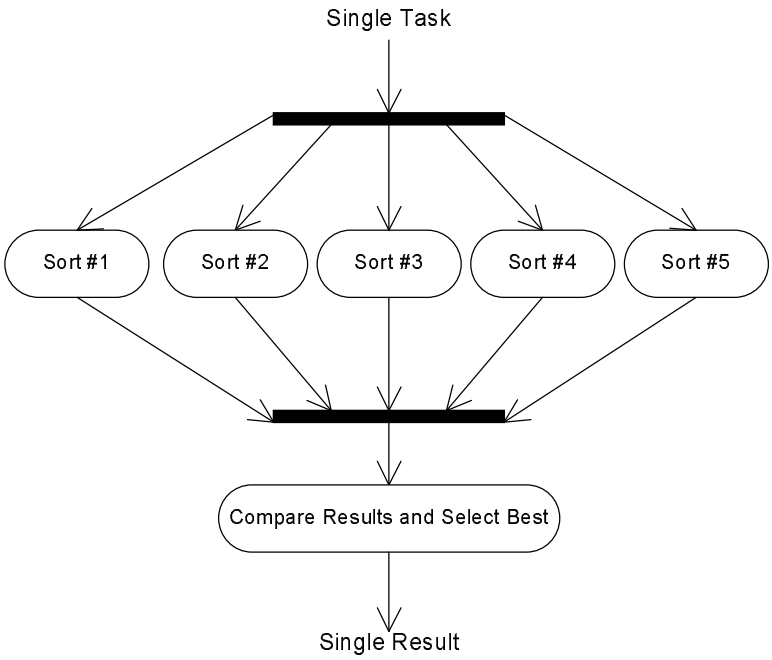

Fig. 7. Centralized architecture for combining N versions of a sorting algorithm into a single, more robust system for sorting, where a postprocessing algorithm chooses one result to be the output

A combination of the preprocessing and postprocessing centralized systems could also be used. Since the characteristics of each module are known, a subgroup could be selected to perform the desired task based on known factors such as speed, time, and space. This subgroup would then have its results compared to determine the best results as above. Because certain modules will be selected every time the same set of circumstances come up, a better way of developing a conventional system would be to hard-wire these sets of circumstances to eliminate the need for a central intelligent filtering unit.

A fourth approach is a distributed solution, where the algorithms jointly decide which one(s) should perform the sorting, and if there is more than one qualified algorithm, they jointly decide on the best result. Conventional algorithms do not typically have such a distributed decision-making ability, so we investigated whether there is a generic capability that can be added to an algo-

rithm to enable it to participate in a distributed decision. We discovered that the result has the characteristics of a software agent.

We collected a number of sorting algorithms, each written by different people and therefore having different characteristics (such as input data type, output data type, and time and space complexity). For our experiments, we converted each algorithm into a sorting agent.

Each sorting agent is composed of a sorting algorithm and a wrapper for that algorithm. The wrapper knows nothing about the inner workings of the algorithm with which it is associated. It has knowledge only about the characteristics of its algorithm, such as the data type(s) it can sort, the data type it produces, its time complexity, and its space complexity. The sorting algorithms were written in Java and the wrappers in JADE [18].

The system sends data to be sorted to all the sorting agents. Their responsibility (as a group) is the sorting of the data, and they should be able to do this better than any one of them alone. Upon receiving data to be sorted, each agent determines whether or not it can sort it successfully (based on the type of the data and its own knowledge of what types it can sort) and if the agent can, it broadcasts a message to every other agent specifying its intention, along with a measure of performance for its algorithm (based on time and space complexity).

The decision of which agent (i.e., algorithm) to choose (among those that are capable of sorting the input data) is made in a distributed manner: upon receiving the estimates from the other agents, each agent compares its own performance measure against those received in the messages. If the agent has the best performance measure, it will run its algorithm and send the results back to the system. If it does not have the best performance measure, it will do nothing. The results, not surprisingly, showed that the agent-based composition of sorting algorithms performed better than any individual algorithm.

7 Conclusion: Challenges and Implications for Developers

Producing robust software has never been easy, distributing it across the Web makes it much more difficult, and the approach recommended here would have major effects on the way that developers construct software systems:

- It is difficult enough to write one algorithm to solve a problem, let alone n algorithms. However, algorithms, in the form of agents, are easier to reuse than when coded conventionally and easier to add to an existing system, because agents are designed to interact with an arbitrary number of other agents.
- Agent organizational specifications need to be developed to take full advantage of redundancy.
- Agents will need to understand how to detect and correct inconsistencies in each other's behavior, without a fixed leader or centralized controller.
- There are problems when the agents either represent or use nonrenewable resources, such as CPU cycles, power, and bandwidth, because they will use it n times as fast.

- Although error-free code will always be important, developers will spend more time on algorithm development and less on debugging, because different algorithms will likely have errors in different places and can cover for each other.
- In some organizations, software development is competitive in that several people might write an algorithm to yield a given functionality, and the "best" algorithm will be selected. Under the approach suggested here, all algorithms would be selected.

Web services are extremely flexible, and a major advantage is that a developer of Web services does not have to know who or what will be using the services being provided. They can be used to tie together the internal information systems of a single company or the interoperational systems of virtual enterprises. But how Web services tie the systems together will be based on technologies being developed for multiagent systems.

Acknowledgements

The US National Science Foundation supported this work under grant number IIS-0083362.

References

1. Berners-Lee, Tim, James Hendler, and Ora Lassila: "The Semantic Web," *Scientific American*, vol. 284, no. 5, May 2001, pp. 34–43.
2. Beugnard, Antoine, Jean-Marc Jezequel, Noel Plouzeau, and Damien Watkins: "Making Components Contract Aware," *IEEE Computer*, Vol. 32, No. 7, July 1999, pp. 38–45.
3. Booch, Grady, James Rumbaugh, and Ivar Jacobson: *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1999.
4. Box, D., et al.: "Simple Object Access Protocol (SOAP) 1.1," 2000. http://www.w3.org/TR/SOAP
5. Coelho, Helder, Luis Antunes, and Luis Moniz: "On Agent Design Rationale." In *Proceedings of the XI Simposio Brasileiro de Inteligencia Artificial (SBIA)*, Fortaleza (Brasil), October 17–21, 1994, pp. 43–58.
6. Cox, Brad J.: "Planning the Software Industrial Revolution." *IEEE Software,* November 1990, pp. 25–33.
7. Decker, Stefan, et al.: "The Semantic Web: The Roles of XML and RDF," *IEEE Internet Computing*, vol. 4, no. 5, September-October 2000, pp. 63–74.
8. Decker, Stefan, P. Mitra, and S. Melnik: "Framework for the Semantic Web: An RDF Tutorial," *IEEE Internet Computing*, vol. 4, no. 6, November-December 2000, pp. 68–73.
9. DeLoach, S.: "Analysis and Design using MaSe and agentTool." In *Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, 2001.

10. Dignum, Frank, Barbara Dunin-Keplicz, and Rineke Verbrugge: "Dialogue in team formation: a formal approach" In van der Hoek, W., Meyer, J. J., and Wittenveen, C., Editors, *ESSLLI99 Workshop: Foundations and applications of collective agent based systems,* (1999).
11. Hasling, John: *Group Discussion and Decision Making,* Thomas Y. Crowell Company, Inc. (1975).
12. Heflin, Jeff and James A. Hendler: "Dynamic Ontologies on the Web," In *Proceedings American Association for Artificial Intelligence (AAAI)*, AAAI Press, Menlo Park, CA, 2000, pp. 443–449.
13. Holderfield, Vance T. and Michael N. Huhns: "A Foundational Analysis of Software Robustness Using Redundant Agent Collaboration." In *Proceedings International Workshop on Agent Technology and Software Engineering*, Erfurt, Germany, October 2002.
14. Huhns, Michael N. and Vance T. Holderfield: "Robust Software," *IEEE Internet Computing*, vol. 6, no. 2, March-April 2002, pp. 80–82.
15. Huhns, Michael N.: "Interaction-Oriented Programming." In *Agent-Oriented Software Engineering,* Paulo Ciancarini and Michael Wooldridge, editors, Springer Verlag, Lecture Notes in AI, Volume 1957, Berlin, pp. 29-44 (2001).
16. Iglesias, C. A., M. Garijo, J. C. Gonzales, and R. Velasco: "Analysis and Design of Multi-Agent Systems using MAS-CommonKADS." In *Proceedings of the AAAI'97 Workshop on agent Theories, Architectures and Languages*, Providence, USA, 1997.
17. Iglesias, C. A., M. Garijo, and J. Gonzalez: "A survey of agent-oriented methodologies." In J. Muller, M. P. Singh, and A. S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98).* Springer-Verlag: Heidelberg, Germany, 1999.
18. *JADE: Java Agent Development Environment,* http://sharon.cselt.it/projects/jade.
19. Jennings, Nick R.: "On Agent-Based Software Engineering" *Artificial Intelligence,* 117 (2) 277–296 (2000).
20. Juan, T., A. Pearce, and L. Sterling: "Extending the Gaia Methodology for Complex Open Systems." In *Proceedings of the 2002 Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 2002.
21. Kalinsky, David: "Design Patterns for High Availability." *Embedded Systems Programming* (August 2002) 24–33.
22. Kinny, David and Michael Georgeff: "Modelling and Design of Multi-Agent Systems," in J.P. Muller, M.J. Wooldridge, and N.R. Jennings, eds., *Intelligent Agents III — Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, Berlin, 1997, pp. 1–20.
23. Kinny, D., M. Georgeff, and A. Rao: "A Methodology and Modeling technique for systems of BDI agents." In *Proceedings of the 7th European workshop on modeling autonomous agents in a multi-agent world*, LNCS 1038, pp. 56–71, Springer-Verlag, Berlin Germany, 1996.
24. Laddaga, Robert: "Creating Robust Software through Self-Adaptation," *IEEE Intelligent Systems*, Vol. 14, No. 3, May/June 1999, pp. 26–29.
25. Light, Donald, Suzanne Keller, and Craig Calhoun: *Sociology* Alfred A. Knopf/New York (1989).
26. Lorge, I. and H. Solomon: "Two models of group behavior in the solution of Eureka-type problems." *Psychometrika* (1955).
27. Nwana, Hyacinth S. and Michael Wooldridge: "Software Agent Technologies." *BT Technology Journal,* 14(4):68-78 (1996).

28. Odell, J., H. Van Dyke Parunak, and Bernhard Bauer: "Extending UML for Agents." In *Proceedings of the Agent-Oriented Information Systems Workshop*, Gerd Wagner, Yves Lesperance, and Eric Yu eds., Austin, TX, 2000.
29. Padgham, L. and M. Winikoff: "Prometheus: A Methodology for Developing Intelligent Agents." In *Proceedings of the Third International Workshop on Agent-Oriented Software Engineering*, at AAMAS 2002. July, 2002, Bologna, Italy.
30. Paulson, Linda Dailey: "Computer System, Heal Thyself," *IEEE Computer*, (August 2002) 20–22.
31. Perini, A., P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos: "A knowledge level software engineering methodology for agent oriented programming." In *Proceedings of Autonomous Agents*, Montreal CA, 2001.
32. Schreiber, A. T., B. J. Wielinga, and J. M. A. W. Van de Velde: "CommonKADS: A comprehensive methodology for KBS development," 1994.
33. Shapley, L. S. and B. Grofman: "Optimizing group judgmental accuracy in the presence of interdependence" *Public Choice,* 43: 329-343 (1984).
34. Swap, Walter C., et al.: *Group Decision Making,* SAGE Publications, Inc., Beverly Hills, London, New York (1984).
35. Tambe, Milind, David V. Pynadath, and Nicolas Chauvat: "Building Dynamic Agent Organizations in Cyberspace," *IEEE Internet Computing*, Vol. 4, No. 2, March/April 2000.
36. Wooldridge, M., N. R. Jennings, and D. Kinny: "The Gaia Methodology for Agent-Oriented Analysis and Design." *Journal of Autonomous Agents and Multi-Agent Systems*, 2000.