Worldwide Computing with Coordinated Adaptive Mobile Web Services

Carlos Varela
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180-3590, U.S.A.
cvarela@cs.rpi.edu

ABSTRACT

Worldwide computing seeks to enable the seamless large-scale utilization of networked heterogeneous resources as a single virtual global computer. Web services use standard internet protocols and markup languages to support the compositional development of portable distributed systems. In this paper, we consider the future of web services as a foundation for developing worldwide computing applications. In particular, we argue that future web service technologies need to provide high-level abstractions for portable dynamic code mobility and decentralized coordination. Code mobility and service coordination are two key critical missing components to add to XML-based solutions for service definition (WSDL), data interchange (SOAP), and service registration and discovery (UDDI).

1. MOTIVATION

XML [4] technologies, developed by the World-Wide Web Consortium (W3C), provide an extensible programming language-independent and application-independent markup language for data. Web services –which use HTTP for communication, and XML for service definition (WSDL), data interchange (SOAP), and service registration and discovery (UDDI)– provide a programming language and device independent methodology for building composable, large-scale, distributed applications.

While XML goes a long way into promoting syntactic interoperability and data and metadata interchange, there are a number of limitations of XML for developing worldwide computing applications:

- **semantic interoperability** –getting a syntactically well-formed XML document does not necessarily imply that there is an agreement on the meaning of the marked up data received from another party.
- **code mobility** –worldwide computing requires highly reconfigurable web services to enable using the web itself as a service execution platform. This, in turn, requires the ability to move code in a highly portable and dynamic manner.
- **inter-service coordination** –building web services by composition requires well-defined coordination models and languages. Coordination includes grouping and composition of web services, automatic service discovery based on service interfaces, different forms of transactional and fault-tolerant behavior, fine-grained resource management, and quality of service.

In the following sections, we define worldwide computing, and present ideas for future web service technologies to enable worldwide computing. In particular, we argue for the need for research in ways of incorporating behavior into XML to support dynamic portable code mobility, and we address the need for scalable web service coordination strategies.

2. WORLDWIDE COMPUTING

The new generation of devices and networking capabilities will extend the Internet into a pervasive and ubiquitous network allowing uniform access to geographically distributed, possibly specialized 'devices' or objects – such as cameras, credit cards, watches, rings and eyeglasses. Such objects can be uniformly accessed and thought of as part of a single, unified *World-Wide Computer* (see Figure 1.)

In this *worldwide computing* vision, traditional assumptions of resource locality and access, such as available memory or data storage capacity, are not meaningful. Software development techniques need to produce code flexible enough to execute over an evolving network. Software will require the ability to be decomposed at run-time and recomposed in different topologies over highly dynamic networks of heterogeneous devices.

If web services are to be used as a fundamental building block for worldwide computing applications, it is critical that they be adaptive to heterogeneous execution platforms,

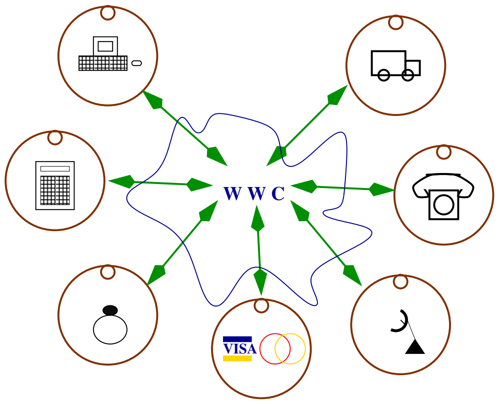

Figure 1: The *World-Wide Computer* (WWC) provides a uniform view of Internet-connected devices ranging from supercomputers and desktops, to cars, phones, smart credit cards and rings.

so that they can be dynamically decomposed and recomposed into multiple hardware devices and software coordination structures.

3. WEB SERVICES MOBILITY

Neither CORBA nor Web Services directly support mobile code, because they assume that communicating entities run on heterogenous hardware and operating systems, and are implemented using different programming languages.

Nonetheless, portable code mobility is critical to deal with system behavior extensibility, practical online deployment of new versions of software, network evolution and dynamicity, limited-resource devices, load balancing, and web service scalability.

The importance of mobile code stems from several perspectives:

- **worldwide computing** –if web services are to view Internet-scale networks as a single global computer, they need to adapt to dynamically changing network topologies and failures by moving code and objects dynamically.
- **dynamic reconfiguration/adaptability** –if web services are to execute on devices ranging from credit cards to supercomputers, they need to be "decomposable" and "recomposable" into different logical connection topologies.

Virtual machine-based systems enable the sharing of byte-code, and promote portability by hiding heterogeneity under the VM. For example, Java applets showed the viability of extending a system's behavior (the web browser) arbitrarily, yet in a secure, encapsulated manner. Moreover, Java's Remote Method Invocation (RMI) includes the ability to dynamically move code across different computers over the Internet by using the virtual machine abstraction layer.

Unfortunately, mobility itself brings along difficulties that need to be tackled. In particular,

- **version control** –mobile web services will need methodologies to deal with different versions of the same code in the network. Old version objects talking to new version objects and viceversa. To accomplish hot swapping [6], adapters need to be written by programmers of new code versions which enable inter-version object communication when meaningful.
- **security** –downloading code from the network in a dynamic manner creates the potential for undesired, potentially detrimental effects on local resources. Web services executing on a VM model can guarantee certain safety properties by wrapping sensitive resource access with security managers enforcing specific site policies (sandbox execution).

Moving code in a portable manner –through a virtual machine abstraction– is a step forward when developing mobile web services. However, beyond byte-code compatibility and therefore portable class mobility, we need mechanisms to support fine-grained object mobility. Moving running instances of a given class is useful to dynamically reconfigure a system –without halting its execution– making it *adaptive*.

Unfortunately, the Java platform[1] provides first-class support for passive code mobility but not for active (running) code mobility. Moving an object in Java during its execution is a very delicate and expensive operation due to Java's passive object model [10]. In Java's object model, threads are free to enter objects, and objects have very limited control on multiple threads that can be active inside an object changing its state [2]. While there is limited support for synchronization [8], it is very hard to move objects across different virtual machines in a generic fashion. Shared memory across multiple objects in the system complicates even further active code migration.

Our research is based on the actor model of concurrent computation in distributed systems[1]. Actors can be thought of as active objects, which completely encapsulate state, procedures to modify the state, and a thread of execution (see Figure 2). We extend actors with a universal naming model [11, 3], and we provide an actor programming language and runtime architecture called SALSA [13].

By providing a universal naming space for actors, we can efficiently accomplish actor migration by serializing an actor's state, behavior, and mailbox of pending messages. Since actors communicate exclusively by asynchronous message passing, it provides a natural distributed memory model immediately prone to migration and systems reconfigurability.

SALSA further enables the development of open, modular, reconfigurable Internet-based systems by providing primi-

[1]We discuss Java since it is the only truly portable virtual machine in existence today, but the principles should also apply to future portable virtual machine specifications and implementations.

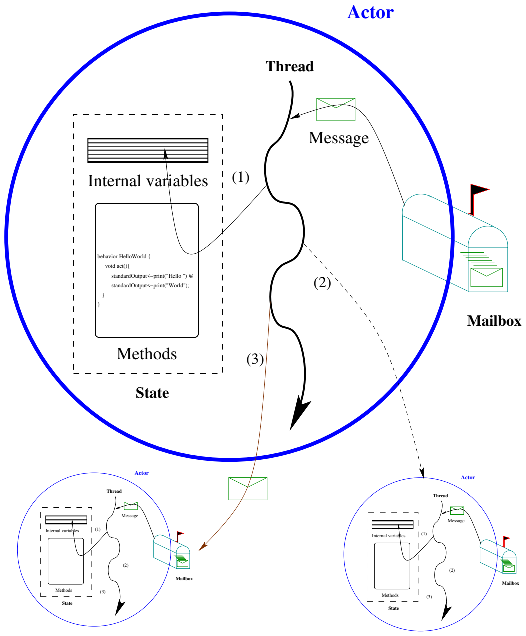

Figure 2: Actors encapsulate state and an execution thread. In response to a message, an actor can change its state, create new actors, or send messages to other actors.

tive coordination mechanisms, including token-passing continuations, join continuations, and delegation. The tight integration between the SALSA programming language and its run-time architecture, the *World-Wide Computer*, provides very high-level programming abstractions for universal naming, migration, and remote communication.

We are using the SALSA and WWC frameworks to further investigate mobility issues, including dynamic reconfigurability, version control, and secure resource management.

4. WEB SERVICES COORDINATION

Coordination models and languages are required to glue together open, modular, heterogeneous web services into meaningful larger systems.

In this section, we introduce two models for coordination based on actors: first, a hierarchical model for coordination which restricts message reception based on actor groups, also known as *casts*, and second, a decentralized coordination model with lazy consistency checking and enforcement designed for building services with different semantic constraints, from best-effort peer-to-peer file sharing systems up to fully transactional electronic commerce systems. Finally, we argue for the need for language support for coordination in web-based worldwide computing.

4.1 Hierarchical Coordination

Varela and Agha defined a model for hierarchical coordination of actors which groups them into casts and designates an actor within the cast as a *director* [12]. The idea is that since all activity in actor systems is triggered in response to message passing, coordination can be achieved by constraining the reception of messages according to the position of the communicating actors in the logical coordination forest of the system.

Under the assumption that communications across system sub-components are orders of magnitude smaller than communications within a component, the hierarchical model of coordination scales up to very large systems. Consider e.g., the Internet's Domain Name System (DNS) [9], where caching and replication ensure that root servers seldom get contacted for information on name lookups.

In the hierarchical model for coordination, messages get automatically routed to the lowest common director between two communicating actors. This scheme ensures that coordination imposes a communication overhead only on actors not closely related in the logical coordination forest.

That is, if two actors are within the same cast, only their common ancestor gets notified about the message sending. It is the director's role to enforce cast coordination policies, such as replication, persistence, storage, or multicasting. If, on the other hand, actors do not have any common director – i.e., they belong to separate trees in the logical coordination forest–, a message will have to visit every director in the way to the target actor. Directors can only send messages to their immediate children in the hierarchy, ensuring local coordination constraints are satisfied. On the other hand, an actor can directly contact any of its ancestors in the coordination forest. We refer the interested reader to [12], for an operational semantics of the hierarchical model for actor coordination, along with some motivating examples.

4.2 Transactors

Field and Varela defined a more loosely coupled (decentralized) model for *transactor* coordination, which enables systems to lazily check for global consistency states in distributed systems [7]. Transactors can *commit* their state to an immutable, persistent one, as long as their state is consistent with all previous communications received from other known actors. Transactors can also *rollback* to an initial state, either spontaneously, or due to failures in the runtime execution substrate. Messages between actors carry a payload to communicate the stability information necessary to enforce consistency in global state at committment time, or to induce rollback behavior for components with inconsistent dependencies.

One of the main design criteria for transactors is to enable interoperation of components providing different semantic guarantees into composite systems providing stronger reliability properties. The idea is that failure modes of different distributed system components should be made *explicit* to enable composite systems to provide extra protocol layers (e.g., retries, replication, logging, rollbacks) if, and only if, necessary to support the desired level of global consistency. With this strategy, high-reliability components –e.g., supporting distributed two-phase commit protocols– can be readily combined with low-reliability components. The price for maintaining strong semantic properties –such as consistent global snapshots, distributed transactions, and so forth– needs to be paid only by applications that truly need them.

Transactors extend and generalize the actor model of computation in two important ways: first, transactors do not assume guaranteed message delivery; and second, transactors provide three primitives for achieving consistent global states, namely, quiescence, commitment, and rollback behavior.

Incorporating failures into the semantic model enables us to reason about unreliable networks and components. We can explicitly determine when messages may not be delivered, or when actors may become unreachable due to node failures or network partitions. Such explicit failure handling enables developing higher-level protocols to provide different levels of semantic guarantees for composite distributed systems.

In addition to the most basic actor primitive operations, namely actor creation, message sending, and internal state change, we incorporate primitives for stabilization, quiescence, and rollback. A stable actor represents an actor with immutable, persistent, and consistent behavior. While a stable actor may not temporarily respond to messages due to node or network failures, its state is guaranteed to eventually be recovered to its last known stable value. The semantics guarantees that an actor is consistent with previous communications from other actors before it can stabilize. That is, other actors which sent messages to the actor in consideration for stabilization, must themselves be stable and must not have rolled back. The *quiesce* primitive lets an actor make a promise of immutability and persistence. It models the first phase in a two-phase commit protocol, and it prevents an actor from spontaneously rolling back.

We refer the interested reader to [7], for an operational semantics of the transactor model along with some motivating examples.

4.3 Coordination Languages

While the hierarchical model and the transactor model for coordination are interesting approaches to deal with the problem of coordination in open distributed systems, there is still a need for research on coordination languages to support these models.

Languages which support a workflow model of coordination generally assume a central point of management and control over the coordinated services or components. Both of these assumptions are too strong in the open world of services advocated by internet computing.

Tuple spaces approaches –originated by Linda[5]– successfully decouple coordinated processes in space and time. However, their main drawback is scalability due to the logical centralization of the tuple space approach.

A realistic coordination language will probably support a hybrid model of coordination, including fully decentralized components with a more structured approach to coordination within the components.

5. CONCLUSIONS

We have discussed what is required of web services as a foundational technology to develop worldwide computing applications, and we have identified two areas for improvement in the current status of web service technologies. In particular, we have considered the inclusion of portable mobile active code into the data interchange offered by XML, and we have discussed two models for web service coordination in open environments. First, a hierarchical model constraining message reception based on coordination forests, actor groups, and group directors. And second, a decentralized coordination model for keeping consistent global states in a lazy manner by piggybacking stability information along with messages. Research is still needed in highly expressive coordination languages to efficiently support different, perhaps hybrid, coordination models in large-scale open dynamic systems.

6. REFERENCES

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha, M. Astley, J. Sheikh, and C. Varela. Modular Heterogeneous System Development: A Critical Analysis of Java. In J. Antonio, editor, *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, pages 144–155. IEEE Computer Society, March 1998. http://osl.cs.uiuc.edu/Papers/HCW98.ps.

[3] G. Agha, N. Jamali, and C. Varela. Agent Naming and Coordination: Actor Based Models and Infrastructures. In A. Ominici, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 9, pages 225–246. Springer-Verlag, Mar. 2001. http://osl.cs.uiuc.edu/Papers/coord01.ps.

[4] J. Bosak. XML, Java, and the Future of the Web. *World Wide Web Journal*, XML Principles, Tools and Techniques, Oct. 1997. See http://www.w3.org/XML/ for more information.

[5] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, 1990.

[6] D. Duggan. Type-based hot swapping of running modules. In *Proceedings of ACM International Conference on Functional Programming (ICFP)*, Florence, Italy, September 2001.

[7] J. Field and C. Varela. Toward a programming model for building reliable systems with distributed state. In A. Brogi and J. M. Jacquet, editors, *Proceedings of the First International Workshop on Coordination Languages and Software Architectures (FOCLASA'2002), affiliated to CONCUR*. Electronic Notes in Theoretical Computer Science, July 2002. http://www.cs.rpi.edu/ cvarela/foclasa2002.pdf.

[8] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley, 1997.

[9] P. Mockapetris. Domain Names - Concepts and Facilities. IETF Internet Draft Standard RFC 1034, November 1987. http://www.ietf.org/rfc/rfc1034.txt.

[10] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in java. In *ASA/MA*, pages 16–28, 2000.

[11] C. Varela. *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign, April 2001.

[12] C. Varela and G. Agha. A Hierarchical Model for Coordination of Concurrent Activities. In P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models (COORDINATION '99)*, LNCS 1594, pages 166–182, Berlin, April 1999. Springer-Verlag. http://osl.cs.uiuc.edu/Papers/Coordination99.ps.

[13] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), Intriguing Technology Track*, October 2001. http://osl.cs.uiuc.edu/Papers/oopsla2001.ps.