Objects as Software Services

Gilad Bracha

August 7, 2006

Abstract

Software services seek to combine the advantages of traditional client applications and web services. Software services go beyond web services in supporting applications that must function when networks are slow, unreliable or nonexistent, that have low tolerance for network latency, and/or require excessive bandwidth. At the same time, software services retain the advantages associated with web services: they are always available and always up to date. We discuss the structure of an object based platform for software services and especially the advantages of object oriented language principles in facilitating the platform's design.

1 Introduction

Web services have important advantages compared to traditional software applications. Briefly summarized, these are:

- *They are always available* (as long as a reliable network connection exists). Users can access them from any node on the internet, irrespective of whether they own the node in question or not, and independent of the peculiarities of specific platforms and operating systems.
- *They are always up to date.* Neither the user nor the developer is concerned with downloading and installing software, or with ensuring that the correct version of it is running.

Web services also have significant drawbacks.

- They require a robust network connection. Otherwise, they cannot be used.
- It is challenging to program such services in a way that provides good interactive response.
- They transfer the entire burden of computation on to servers, making poor use of the computational capacity of devices on the network edge.

Software services are designed to address the weaknesses of web services while retaining their primary strengths. In this paper, we sketch an object based platform for software services. The emphasis in this presentation is on the leverage our project gains by adhering to specific principles of object oriented programming language design.

The prominence of web services has already provoked significant efforts in object oriented language design, in two areas:

- Support for semi-structured datatypes. Web services traffic heavily in XML, and language support for representing, manipulating and typechecking XML data has received a great deal of attention [BMS05].
- Support for asynchronous distributed computation [BCF04].

While a language should offer support in these important areas as well, this paper focuses on issues that arise largely from the needs of software services, which have not received much discussion in the literature.

Software services must be usable off-line (i.e., in the absence of a working network connection). This implies that the service must be able to run on a client device and that the data the service requires should be available on the client. At the same time, when a network connection is available, data must be placed on a network accessible server, so that it can be made available to other clients, reliably backed up etc.

Since multiple clients might edit the data while off line, a reconcliation process must ensure that the server retains a correct, up-to-date data representation. This process is usually known as *synchronization* and is commonplace in applications such as personal information managers (PIMs). Usually, this requires specialized programming on the part of each application. Furthermore, such data synchronization in no way addresses the management of program downloads, versioning and updates that web services handle so well.

Our design addresses these two key points with the notions of:

- Orthogonal Synchronizability.
- Programs as Data.

These notions, while based on a long history of computing research, are undoubtedly controversial. In the next two sections, we discuss them in detail, and argue why they are appropriate in the context of software services. Then, we discuss issues of security and concurrency their interaction with language and platform design. Finally, we present related work and conclude.

2 Orthogonal Synchronizability

Orthogonal synchronizability is a novel contribution of this work. A datum is *synchronizable* if it can be reconciled with a corresponding (possibly different or even empty) datum stored separately (on a server or on a different client device). Orthogonal synchronizability means that any datum, regardless of type, is synchronizable.

In practice, this means that any object in an application can be correctly synchronized, without special effort on the part of the programmer, This enables a wide variety of applications to be programmed more naturally and robustly while still supporting the synchronization features critical to a software service.

Readers will no doubt recognize the similarity to the well known notion of *orthogonal persistence* [AM95]. Orthogonal persistence has fallen from favor. This stems largely from the fact that data created by orthogonal persistence tends to be in the form dictated by the program. This has some inherent disadvantages:

1. Data typically outlives the program that created it. Such data is at risk of becoming inaccessible as the tools that create it become obsolete and eventually unavailable on current platforms.

2. Programs represent data in ways that are optimized for the program's needs. They tend to include transient fields that are not properly part of the data's natural representation.

3. Data needs to be shared by multiple programs with different needs. Using a representation dictated by one program can make such sharing difficult.

We believe that these concerns, while valid in their original context, do not pose a problem in the new world of software services.

Software services are responsible for both data and applications that manipulate it, by definition. As long as the service exists, a program that understands how to manipulate it exists as well. Indeed, the data representation and the applications that manipulate it evolve in tandem. If the service ceases operation, the data must be extricated at that point in time, regardless of its format. Thus, item 1 no longer holds.

Transient data is already filtered out of the synchronized representation, as discussed in subsection 2.3 below, so we need not be concerned over item 2.

Likewise, point 3 is moot. Web services naturally traffic in relatively stable, standard representations of data (as XML) and therefore such applications provide ways to exchange their data with other applications.

On the other hand, the benefits associated with orthogonal persistence carry over to orthogonal synchronizability. Programmers are relieved from a heavy burden of design and implementation to support synchronization. Even simple, small programs can act like a service, without the heavy overhead of supporting a special infrastructure. The resulting system is more robust and performant, as the specialized effort involved in implementing synchronization is centralized.

Now it is time to show how orthogonal synchronization is implemented in practice.

2.1 Synchronization Scheme

Our synchronization scheme is change-based. When synchronization occurs, the client relays a change log to the server. The server then applies these changes to the last version known to exist on the client (known as an *anchor*), to derive a complete current representation of the client's state. This version is then reconciled with the current master version to derive a new master version. Finally, the set of differences between the newly computed master version and the version on the device is computed and sent to the client, which applies it locally.

This scheme contrasts with approaches such as Harmony [PSG04], which seek to reconcile complete data sets.

The advantage of a change-based scheme (in many ways similar to SyncML) is that it conserves bandwidth, enabling quick and more frequent syncs. Short syncs save not only time, but device battery power (in the common case where the client is a mobile device).

Furthermore, we elect to place most of the burden of reconciliation on the server. Reconcliation is computationally demanding in both time and space. Typically, reconciliation requires at least twice the memory footprint of the data set, as two copies of data need to be stored and compared. The savings in space, time and, once again, battery power on clients are crucial, especially for small mobile devices.

All of the above begs a key question: How is the change set derived on the client?

2.2 Maintaining Change Sets on the Client

Using conventional techniques, maintaining an accurate change log is difficult. Any data that must be synchronized would need to log changes to itself as the computation occurs, This would require error-prone manual programming. Furthermore, the set of data reachable from a persistent (or in our case, synchronizable) root changes dynamically. When do we log these changes, and when do we not?

Our solution relies on the system's ability to dynamically modify objects so that they begin to log changes when they become reachable from the set of synchronizable roots. To do this, we require that all state be accessed via accessor methods. The use of accessors is good software engineering practice, and is easily enforced by the language. In this respect, we follow the lead of Self [US87].

Our system is capable of dynamically changing these accessor methods to log (or stop logging) changes. Thus, the ability to modify code on the fly is an intrinsic requirement in our scheme. Of course, such a capability brings with it potential security risks. We address these using a mirror based reflective architecture [BU04] coupled with capability based security, as discussed in section 4.1 below.

2.3 Handling Transient Data

Transient data is purged automatically upon synchronization. This enforces a scheme in which applications ensure all accessors for transient data compute that data lazily if it is nil.

In this scheme, all transient fields are associated with a closure which computes their value. This association is enforced syntactically. Whenever the value of a transient field is required, the system generated accessor will check if the field is nil. If so, it will initialize it using its associated closure and return the computed result. Subsequent requests use the cached value. Niling out the transient fields therefore force all transient data to be lazily recomputed. Lazy initialization of all transient values is thus assured, both initially and after every synchronization. This means that changes to persistent data as a result of a sync are always reflected in the program's transient state.

3 Programs as Data

Treating programs as data means that all the benefits of synchronization carry over directly to programs as well. Just as there is one current version of the data (with auditable, consistent snapshots representing the past), there is one current version of all applications, which is consistent with the data (and likewise, consistent snapshots of the past).

This economy of mechanism translates directly to a smaller platform, which has smaller footprint and is easier to maintain.

Nevertheless, some care must be taken for this vision to work.

3.1 Reflection

As noted above, during synchronization, clients send a change set to the server. The server then updates the master version, computes what changes are needed to the client, and sends these changes back to the client. The client then applies these changes to itself.

Since changes may include modifications to the program as well as data, the client must be able to modify the program. To prevent any disruption of the user's work, these modifications must be applied without stopping the running program.

It follows that reflection, and especially support for self modification (also known as *hotswapping*), is crucial so that running applications can be updated during synchronization.

The ability to modify code on-the-fly is also used to support orthogonal synchronizability as discussed above.

Modifying a running applications is a delicate operation. While modifying the code is straightforward, modifying the structure of objects is more complex. However, these issues have long been addressed in programming environments such as Smalltalk [GR83], Self [US87], Lisp [KdRB91], Erlang [AVWW96] etc.

When modifying a running application in a production environment, special considerations apply. If the code to be modified is active on the call stack, what should its disposition be? The answer depends on circumstances. In our case, the system ensures that this circumstance does not arise. All applications must be quiescent when synchronization is invoked. This fits well with user expectations, since synchronization provides a natural hiatus in user interaction.

The requirement that applications be quiescent ensures that their call stacks are empty. A quiescent application is typically waiting for an input event. As such, it has no active frames on its call stack (except possibly the event loop itself). It should be clear that there is no need to stop and restart an application. Following synchronization, all the persistent state being acted upon by an application is in memory and up to date. Any transient data that the application requires will be recreated on demand as noted above.

3.1.1 Typing

From a programming language design perspective, the most crucial property needed to enable reflective update is dynamic or optional typing.

In the presence of mandatory static typing, reflective update requires the run time system to maintain the invariants of the static type system in the face of program changes. Naively, this would require retypechecking the entire system, a costly proposition. This may be optimized using incremental typechecking, but this remains resource intensive: dependencies must be maintained, and the amount of code to be revisited may still be large. This explains why systems that rely on mandatory typing (e.g., Java [GJSB05]) do not support the flexible reflective features we require. Type safe hotswapping remains, therefore, a topic for further research [Dug01, SHB$^+$05].

Note that we do not advocate the complete absence of static typechecking, as in traditional dynamically typed languages. On the contrary, static typing can have great value in program development. The important thing is rather that typing is not mandatory, but instead is optional and even pluggable [Bra03, Bra04]. Pluggable typing also allows for the seamless incorporation of specialized type systems that support security - for example, information flow tracking types that ensure that capabilities do not escape their intended scope.

3.2 Modularity

Programs are installed onto various devices, where they interact with facilities of the host. It is therefore essential that any external dependencies of a program be clearly distinguished.

A module, therefore, is completely self contained. All external references are explicitly defined as parameters to the module, and their bindings are determined only when the module is used.

A module is stateless. It may be instantiated to an object, at which time actual parameters are provided, binding the module to its environment.

There is therefore no global or static state in our language. The only global names denote immutable values.

Namespaces map names to immutable values. Namespaces are themselves immutable, and may therefore nest. There is a global namespace whose structure is similar to that of Java packages - an inversion of the internet domain system.

Namespaces may be self contained, so that definitions within them do not have access to any surrounding namespace. We provide operators, in the style of Jigsaw [Bra92] that allow a namespace to restrict the names that it exposes to the outside. A module is then a parametric, self contained namespace with an explicit list of exposed elements. Only the module parameter declarations have access to the surrounding namespace, so that they may be annotated with metadata defined elsewhere - in particular, types. Any access to types within a module must be be through a parameter name.

Stateless modules provide advantages with respect to:

- Distribution. Modules can easily be shared across processes, both local and remote.
- Security. Static state provides *ambient authority*, and as such is a well known source of security problems.
- Memory management. Modules can be unloaded from memory through ordinary garbage collection. They do not necessarily need to be stored in a special memory space, improving memory utilization. All this simplifies garbage collection. It is interesting to contrast this with the Java platform, where class unloading is restricted and subtle [GJSB05].
- Startup time. In the absence of static state, programs are much less likely to indulge in excessive initialization up front, and more likely to build up state lazily. This allows for faster startup and a better user experience, which is important in a client system.
- Robustness. Since classes need not be initialized, class initialization cannot deadlock, unlike mainstream programming languages [GJSB05].

Our model fits well with security needs. Each module instance has its own sandbox, tailored to its requirements based on the principle of least authority (POLA). This provides both tighter control and greatly increased flexibility compared to a traditional sandbox model.

When a module is instantiated on a host, the objects provided as parameters define the sandbox for that module. These objects are capabilities, as described below.

4 Capability based Security

Work on mobile code security is focused on protecting the host computer from mobile code. The flip side of this concern is protecting the code from unauthorized access by the host. That is the domain of licensing and digital rights management. Capabilities work well for both concerns. Again, this conceptual parsimony yields reduced software size, complexity and cost throughout the software lifecycle.

4.1 Mirrors as Reflective Capabilities

The capability to perform reflection subsumes any other capability in the system, as it enables arbitrary data access and program change. Therefore, it is crucial that this capability be managed very carefully. In most reflective APIs, access to reflective functionality is diffuse throughout the system.

In contrast, a mirror-based reflective architecture [BU04] provides a single access point for all reflective facilities. In our design, this is manifested as a singleton instance of the Mirror module. This instance is the factory for all mirrors in the system. As such, it is the sole capability for reflective access. More restricted capabilities may be provided by the mirrors it produces, or by wrapper objects that restrict its functionality.

4.2 Subscription based Licensing

The model described here naturally enables subscription based services. Any software module can be disabled when/if its license expires, and arbitrary licensing schemes can be implemented - e.g., an object that only plays a given piece of music once, or five times, or until a given date. Of course, the ability to disable a given module is itself a capability that must be managed carefully.

Subscription based licensing is a form of digital rights management. The challenges in this situation are different from those that arise in securing the platform from mobile code. Here, the owner of the host machine is a potential adversary, and this owner has a great deal more power at his/her disposal than incoming mobile code has.

In principle, the platform can be reverse engineered, subverting any licensing scheme it implements, This is, however, true of any software DRM scheme. In practice, if there are reasonable incentives to use the system legally and the system is well designed, this is not a large problem.

There are nevertheless issues to consider. DRM schemes are best protected if they are directly supported by the OS. Running a platform directly on the hardware, as its own OS, is attractive. If this is not feasible, cooperation/support from the OS is helpful. A third possibility arises when the host platform is itself managed, e.g., a mobile phone, where the telco controls all downloads to the device.

In many situations, none of the above options is available. In this case, one must be very careful in controlling access to native code. Access to native code is the ultimate capability. Unrestricted calls into native code afford that code the opportunity to undermine any guarantees the platform can make. It may be necessary to ensure that access is only granted to specific, well known native libraries (through platform primitives) and/or across process boundaries (see section 5). Even in the case of RPC, one must be very careful what structures are passed across the process boundary, so no critical data, such as cryptographic keys, is inadvertently exposed.

5 Actor-style Concurrency

Web services are distributed processes that communicate asynchronously. The software service platform's notion of concurrency is likewise based on isolated processes known as actors, that communicate by asynchronous message passing. Only immutable data is shared among actors.

The concept of immutable data is general use in a programming language, and helps to efficiently model mathematical values such as numbers and booleans as objects. As we have seen above, it also applies to program constructs such as namespaces, modules and classes. This very same construct also facilitates communication among actors.

The actor model fits well with the GUI/Logic/Database model used in web applications today. It naturally encourages this structure in applications regardless of whether they are actually distributed in this way or not. At the same time, it avoids the forced coding of each component in a different language.

As a result, programmers need not worry about how to split their application between client and server. This concern is one of the fundamental architectural difficulties with current approaches to web programming such as AJAX. Actors can be migrated transparently from client to server or vice versa. Code that cannot run on the server today for whatever reason (network performance, server resources) can be moved to the server as conditions change. The migration can even be done dynamically.

Actors don't know if their peers are running on the same VM, the same machine or even if they are actual actors or just proxies for processes written in other languages on other platforms. For example, a C or Java process can communicate with an actor via the system's asynchronous message passing protocol. The external process appears to be just another actor - regardless of whether it a web service on a remote device or a local process written in some other language.

Just as a web service can masquerade as an actor, it is easy to adapt an actor to be a web service in this model. A standard mapping from asynchronous method invocations to an HTTP or SOAP protocol can be implemented by an adaptor process which can be generated reflectively on the fly.

As alluded to in section 4.2 above, the actor model provides a way to communicate with native code without the security risks that usually entails. Recall that in Java, any security guarantees made by the JVM may be voided by misbehaving native code. Such code might have buffer overflows that corrupt the VMs memory; malicious native code might snoop on VM memory and defeat encapsulation and privacy policies supported by the platform. If native calls are restricted to interprocess communication, these risks are greatly reduced. Of course, there is a significant performance penalty for such an arrangement.

The actor model also has significant advantages for programmers. Concurrency is easier to reason about in this model, because there is no shared state.

6 Advantages of Software Service Vendors

A significant advantage of web services is that their disposition can be constantly monitored by the service provider. When a web service fails, the service provider knows that it has failed. If the service provider and the software vendor are one and the same, or in close contact, the vendor can attempt to diagnose the problem. The service provider also knows what configuration of software is in use; again, as long as there is a close connection between provider and vendor, there is no concern for unforseen configurations that might be deployed in the field.

Software services can ensure that the developers of software always enjoy these advantages. Every time a program fails, the system can log data about the failure and ensure that it is reported. Once a reliable fix is available, it is deployed automatically to all users.

However, software services can go beyond web services to support robust software evolution. The service model works well for applications. However, in the case of software libraries, versionless evolution is harder. Subtle incompatibilities may break client code. The traditional solution is to allow clients to retain old versions of libraries for as long as is convenient.

Of course, one can simply rename classes when an incompatible change is found, and keep both versions available. This again has implications for language design. Mandatory type systems may inhibit such scenarios.

However, the goal has been to eliminate software versions altogether. The world of software services brings a new mindset in terms of software maintenance and testing.

Since every use of the software is known and recorded on the service's servers, library vendors can release updates at a very high frequency and the release cycle is effectively shortened from years to weeks or days. Clients would get early versions and test them (in fact, the service could run tests suites automatically) and make adjustments (or demand adjustments from library vendors) on the same rapid time scale.

Every configuration is known. Vendors can know with certainty who is using a deprecated method or class. They can contact these people, or even provide automatic updates that replace one usage with an equivalent, more up to date one.

7 Related Work

Efforts to combine some of the advantages of web services and of traditional applications have been concentrated in the commercial world. These include systems such as DreamFactory, Flash based systems like Laszlo [Las], Macromedia Flex & Central , and Water [Plu03]. Most of these systems are focused on providing a rich client UI experience while supporting network aware applications. Only Central provides a deployment model, and none have provided a mechanism for updating programs on the fly. Hence, the fundamental move toward truly versionless applications is not addressed by these systems. Security is also typically lacking.

The E programming language [Sti00] focuses on issues of security and distribution, and its designers have made similar decisions. Like our work, it is heavily influenced by the legacy of Smalltalk-80 [GR83]. However, in the interest of security, E completely eschews reflection, which we have shown to be crucial to our purposes. E also does not deal with the notion of software services in any particular way.

One of the commonalities between our system and E is the absence of static state. Fortress [ACL$^+$05] also restricts static state, and has a component/module model that is not dissimilar to our proposal. Recent versions of Scala [Ode01, OZ05] also ban static state.

There is a vast literature on modularity, and we cannot begin to survey it here. Notably relevant is the work on Units [FF98a, FF98b]. ML modules [Mac84, MTH90] are similarly parametric, but differ in crucial respects - they are not first class values and they do not support mutual recursion.

Harmony [PSG04] is a system for synchronization that is not based on change sets, but allows for synchronization of heterogenous data sources.

There are numerous commercial standards that deal with some of the issues we discuss. SyncML gives a model of changed based synchronization that is similar to ours, It does not however provide for orthogonal synchronizability, or deal with the synchronization of programs. It is, however, programming language independent. WebDAV is a standard for file systems access over the internet.

8 Conclusions

We have outlined a platform for software services that combines advantages of web services and traditional client applications.

Contributions of this paper include

- The novel concept of orthogonal synchronizability. While building upon established notions of orthogonal persistence, it is better suited to practical application, taking advantage of the special circumstances of software services.
- The reification of programs as *stateless* data in an object oriented language, and the direct application of this idea to version-free application deployment.
- The notion of a general purpose language (and underlying VM) that directly support capabilities.

References

[ACL$^+$05] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy Steele, and Sam Tobin-Hochstadt. The Fortress langugae specification, 2005. available from http://research.sun.com/projects/plrg/.

[AM95] Malcolm P. Atkinson and Ronald Morrison. Orthogonally persistent object systems. *VLDB J.*, 4(3):319–401, 1995.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang, Second Edition.* Prentice Hall, 1996.

[BCF04] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[BMS05] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in c*mega*. In *ECOOP*, pages 287–311, 2005.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

[Bra03] Gilad Bracha. Pluggable types, March 2003. Colloquium at Aarhus University. Slides available at http://bracha.org/pluggable-types.pdf.

[Bra04] Gilad Bracha. Pluggable type systems, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 2004.

[Dug01] Dominic Duggan. Type-based hot swapping of running modules. In *ICFP*, pages 62–73, 2001.

[FF98a] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[FF98b] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley, Reading, Massachusetts, 2005.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[Las] http://www.laszlosystems.com/.

[Mac84] David MacQueen. Modules for Standard ML. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pages 198–207, August 1984.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Ode01] Martin Odersky. Report on the programming language Scala, 2001. available at http://lampwww.epfl.ch/ odersky/scala/.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 41–57, 2005.

[Plu03] Mike Plusch. *Water: Simplified Web Services and XML Programming.* Wiley Publishing Inc., 2003.

[PSG04] Benjamin C. Pierce, Alan Schmidt, and Michael B. Greenwald. Bringing Harmony to optimism: An experiemnt in synchronizing heterogeneous tree-structured data. Technical Report MS-CIS-03-42, Department of Computer and Information Science, University of Pennsylvania, 2004.

[SHB$^+$05] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 183–194, January 2005.

[Sti00] Mark Stiegler. E in a walnut, 2000. available at http://www.skyhunter.com/marcs/ewalnut.html, or from http://www.erights.org/.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, pages 227–242, December 1987. Published as Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, number 12.