Research Statement

Krzysztof Ostrowski

1 Overview

The goal of my research is to develop programming abstractions that facilitate building scalable Web applications. I believe that scalability should be implicit and automated by a compiler or a runtime environment, similarly to how we automate, e.g., garbage collection in Java or register allocation in C. I would like to provide abstractions that leave maximum flexibility to a compiler in deciding how to store and update the state and content of a Web application, where to deploy its constituent parts, and how they should communicate. Towards this broader goal, my research so far has focused specifically on distributed multi-party protocols as means of storing/updating Web content. I explored architectures, in which Web clients at the edge could freely communicate not only with servers in the cloud, but also directly with one-another, in a P2P fashion. I believe that the key to scalability on the Web is to go beyond the client-server paradigm, and exploit the full spectrum of communication/storage technologies, e.g., P2P overlays, IP multicast, gossip, and replication. The chosen mechanism should reflect the type of Web content, and take a different form in different regions of the network, where the physical architecture, churn, or communication patterns could vary significantly.

In publications for the Web research communities [1, 2, 3, 4, 5, 6, 7, 8], I argued that this type of flexibility is at odds with the principles that underpin the existing Web technologies. Consequently, I adopted a clean-slate approach. My contributions are briefly summarized below, and discussed in more detail in Section 2.

1. In my Ph.D. dissertation, I have described Live Distributed Objects (LDO) [4, 9], a new object-oriented programming model for the Web designed from ground-up with architectural flexibility in mind (Section 2.1). LDO decouples application logic from the underlying communication/storage protocols in a way that is entirely protocol-agnostic. To demonstrate the usefulness of the model, I have developed a platform [10], in which one can compose interactive, content-rich mashups and shared workspaces reminiscent of Google Wave and Chrome OS[1]. Its unique feature is that it can store content not only in the cloud, but also on the clients, using P2P replication or even customized protocols developed by the user: the underlying communication/storage infrastructure can be declared as a part of a mashup.

2. To facilitate formal reasoning about systems built with LDO, I created Distributed Data Flows (DDF) [11, 12, 13], a new distributed system formalism and a programming language (Section 2.2). Whereas LDO is an extension of the object-oriented programming paradigm, DDF equips it with functional semantics. Its unique feature is that it allows hierarchical systems to be modeled as recursive programs. It creates a new type of flexibility for a compiler to decide how to build a scalable hierarchy, aggregate or disseminate data in different regions of the network, without affecting the semantics of a protocol.

3. The ideas that underlie LDO/DDF evolved as generalizations of practical experiences building QuickSilver Scalable Multicast (QSM) [14, 15], a reliable multicast platform for enterprise LANs (Section 2.3).

4. To demonstrate the versatility of LDO, I created Self-Replicating Objects (SRO) [16], a new concurrency abstraction based on it (Section 2.4). SRO can automatically parallelize sequential code to exploit multicore CPUs. It is backwards compatible with the Actor model style of concurrency, and extends it by parallelizing individual Actors without the need for synchronization primitives. In doing so, SRO seamlessly combines the benefits of multiple paradigms: Actors, data parallelism, and MapReduce.

[1] LDO can also be used to build 3D virtual worlds or multiplayer games, as shown in streaming videos on the project website [10].

2 Summary of Key Results

2.1 Live Distributed Objects (LDO)

LDO is based on a vision of the Internet as a single global runtime environment, an analogue to Java JVM or .NET CLR, and distributed multi-party computations as first-class objects created to run in this environment [4, 9]. Unlike in Java, a single object in LDO is a collective entity that can span across a dynamically changing set of locations, e.g., a distributed lock or transaction, a shared document, or a multicast stream. By adopting this bird's eye perspective, we can express hierarchical distributed systems as recursive data structures [13]. This enables us to apply established high-level programming language techniques in completely new ways, to formally express and reason about hierarchical relationships in large heterogeneous systems (Section 2.2).

As a component integration technology, LDO generalizes web services; it provides strong, language agnostic decoupling between components, it supports hierarchical composition, and it subsumes web services as a special case. Unlike in web services, composition in LDO occurs on the clients [5]; mashups expressed in LDO markup language are assembled on the fly, in the user's browser. LDO relies on structural conformance and automatically generated wrappers to match/connect client-side artifacts that need to interact, but that have been built independently, were not expected to work together, and share no API in common. Such late-binding will likely become a dominant model on the Web as mashups get more collaborative, let users drag and drop custom designed interactive objects and connect them to one-another in nontrivial ways [5].

LDO introduces a new architecture-agnostic storage abstraction, a Checkpointed Channel (CC) [8]. CCs generalize the concept of ordinary variables. A CC is a stream of state checkpoints and incremental updates. It can encapsulate its content internally, or it can store all content on the clients, and act merely as a medium that orchestrates the exchange of checkpoints and updates; the decision can be made at runtime, and it does not affect the API used by the application. CCs can store structured data, including references to other CCs. A typical LDO application is a graph of hyperlinked CCs, each storing a part of a hierarchical UI, potentially using a different protocol adjusted to the type of its content [6, 7]. By abstracting the storage protocols, CCs enable hybrid storage systems that span across the edge and involve the cloud and its clients (Section 3.2).

2.2 Distributed Data Flows (DDF)

Just as LDO introduced a bird's eye view on the hierarchical structure of a distributed system, DDF provides a bird's eye view on the interactions between its components. Since each object in LDO spans a dynamically changing set of nodes, so does the history of interactions between any pair of objects. In the DDF language, a single value represents a stream of events distributed in space (across the network) and in time [11, 12, 13]. Each object can be modeled as a function that transforms such values; a hierarchical system is then modeled as a recursive function. Just as in a functional language the runtime has the freedom to make recursive calls in parallel, in DDF the runtime can choose how to build a scalable hierarchy; the decision will not affect the overall semantics of the application. This opens up an intriguing possibility: we can think of heterogeneous communication/storage architectures as results of automatic compiler/runtime optimizations (Section 3.1).

I designed a programming language to support this model [12, 13], an architecture onto which programs expressed in this language can be compiled [1, 2, 3, 11], and a theory that facilitates formal reasoning about these programs [11]. Many simple protocols can already be expressed in the DDF language [12, 13], e.g., distributed locking, leader election, reliable multicast, and various forms of agreement. I am actively working on refining the theory and language to expand the set of protocols that can be expressed and proven correct.

As an architecture, DDF lies in a sweet spot between loosely-coupled P2P overlays and group communication systems. Like the former, DDF incurs $\mathcal{O}(1)$ average and $\mathcal{O}(\log n)$ maximal overhead in any part of an n-node system [11]. Like the latter, it can provably [11] make irreversible decisions and remember them. DDF achieves this by combining elements of the two approaches. First, a hierarchy of membership services is established that mirrors the hierarchy of administrative domains, each managing a portion of the Web; a hierarchy of P2P protocols is then laid over this backbone infrastructure to do the actual work, each protocol managed by a corresponding membership service. Such architectures, using an infrastructure rooted in the cloud to organize a set of clients, are likely to be the next evolutionary step in cloud computing (Section 3.2).

2.3 QuickSilver Scalable Multicast (QSM)

QSM is a reliable multicast engine for enterprise LANs that uses IP multicast to disseminate data and a P2P overlay for loss recovery. Such systems tend to be unstable; reactive recovery mechanisms often trigger too late and lead to a disproportionate response. In building QSM, the challenge was to understand the nature of this instability, and find a way to alleviate it. The work uncovered strong connections between the global performance of the system and purely local phenomena, such as scheduling or memory overheads [14, 15]. QSM introduces a variety of techniques to better tolerate local perturbations and to reduce the staleness of the control information it acts upon, e.g., priority handling of control traffic using a custom scheduler, and pull-based protocol stack. Although implemented in C#, QSM can nearly saturate the underlying network with very low CPU footprint, and can scale from 2 to 200 nodes with less than 5% performance degradation.

The recent trend to consolidate resources in data centers to save energy means that systems will increasingly run at close to their maximum capacity, and will be increasingly prone to instabilities. This is precisely the challenge that work on QSM addressed; some of my findings may be applicable in this broader context.

2.4 Self-Replicating Objects (SRO)

As a programming abstraction, SRO is based on LDO, in that a single component in SRO represents a group of replicas working together. The difference is that in SRO, the replicas usually reside on the same machine; they run in parallel on different CPU cores. Externally, an SRO component resembles an ordinary object; it can define any number of methods. Internally, it can spontaneously and transparently partition its state, to be able to handle multiple method calls in parallel. Each of its replicas handles a disjoint subset of the calls, one at a time, and each replica works on a different portion of the component's state; hence, there is no need for synchronization; all aspects of synchronization, scheduling, and replication are handled by the runtime. For calls that cannot run in parallel, the runtime automatically merges all replicas prior to execution. During its lifetime, a component can undergo many replicate/merge cycles in a manner reminiscent of MapReduce; the runtime automatically adjusts the number of replicas in response to the workload. At different moments during its lifetime, an SRO component can behave like an Erlang-style Actor, it can implement MapReduce or other forms of data/task parallelism; developers need not commit to a specific paradigm in advance [16].

3 Directions of Future Work

3.1 High-Level Programming Languages for Specifying Distributed Architectures

Existing programming languages and tools provide very limited support for modeling distributed systems; hierarchical architectures or those that involve distributed multi-party protocols tend to be presented informally, in the form of diagrams. I would like to evolve LDO/DDF into a mature programming methodology that allows any type of a distributed architecture to be compactly expressed in a form that is easy to analyze by a human, but that can also be automatically processed by a compiler and transformed into code. I envision that constructing a complex distributed architecture could resemble building a Java/.NET application, and involve elements such as object relationship diagrams, type checking, or intellisense. The development environment would be able to, e.g., suggest a multicast stream vs. a gossip protocol as a connector between components, or a replicated state machine vs. a hosted web service as a form of storage, and statically check whether the system as a whole is correct. LDO/DDF is a step in this direction, but it can express and verify a limited set of protocols, the language lacks a type system and a full denotational semantics for data flows.

Ultimately, I would like to shift as much low-level coding burden as possible to a compiler, as postulated in Section 1. Thus, in contrast to formalisms such as TLA or I/O automata, my focus is not on detailed, low-level specification of specific systems, but rather on high-level expression of a system architect's intentions; concrete systems would be generated by a compiler, using a library of micro-protocols and design patterns. Besides completing the LDO/DDF language, I plan to develop a family of compiler optimizations that will improve the quality of the generated distributed architectures, e.g., by avoiding redundant processing [13].

3.2 The Clients + the Cloud: Hybrid Architectures for Offloading Hosted Services

Migrating applications into the cloud offers many benefits, such as security, persistence, or predictable QoS, but it has disadvantages; thin client architectures do not fully exploit the processing power, storage capacity, and communication bandwidth available on the client machines. The industry is turning to hybrid *clients + the cloud* architectures [17] that could exploit scalable P2P mechanisms as an extension of the cloud services. In general, I am interested in exploring the ways in which a scalable P2P infrastructure formed by clients at the edge can boost the performance of a hosted service, e.g., by removing a bottleneck or lowering resource footprint at a data center, and in adaptive solutions that can do it automatically, e.g., based on workload. In this context, there are two areas I would like to target: *managed P2P* systems that self-organize a set of clients into a reliable fault- and churn-tolerant service under control of the cloud, and *hybrid services* that combine cloud and P2P services into a single abstraction to combine their QoS properties. These are discussed below.

In a managed P2P system, a small set of services in the cloud serves as a foundation, upon which clients dynamically construct a hierarchical overlay or other scalable structure. Each level depends on levels above it as a reliable source of consistent configuration, with hosted cloud services at the root indirectly managing the entire system. Each level uses replication or other similar technology to provide consistent configuration to levels below it, e.g., the clients might dynamically bootstrap a hierarchy of replicated state machines that act as membership services in a hierarchical reliable multicast protocol of the sort used in DDF [1, 2, 3, 11]. I am interested in the types of scalable architectures that can be built in this manner, what strong guarantees can be made about these architectures, and how these guarantees can be exploited to build reliable services.

A hybrid service combines multiple existing mechanisms, e.g., a hosted web service and a managed P2P service, and either uses them both simultaneously or switches between them in a coordinated fashion based on the workload. To the application using it, such mechanism appears as a single service that combines the QoS guarantees of the underlying mechanisms. In this context, I am especially interested in hybrid storage substrates that combine scalability with persistence. For example, a group of clients might store content in the cloud when their number is small or when churn is high, but orchestrate a coordinated switch to a P2P mechanism when the metrics cross a certain threshold; by doing so, one could maintain a desired minimum level of persistence, yet avoid overloading the cloud service. I am interested in exploring what types of QoS guarantees can be efficiently combined, and how to design the supporting adaptive switching mechanisms.

3.3 Programming Languages for Secure Collaborative Mashups of Data and Code

Documents are generally considered different from programs, but as Web mashups get more sophisticated, involve structured data, control flow, and complex content pipelines, this distinction becomes less obvious.

Tomorrow's mashups will enable users to overlay, filter, combine, or otherwise process live data streams in nontrivial ways. I have argued [5] that to facilitate the sorts of architectural flexibility postulated in Section 1, the mashups will need to run on the clients in the browser, and declare their underlying communication and storage stacks much in the same way web pages today declare their hierarchical UIs. In this sense, mashups are becoming very much like Java programs, and will need to inherit some of the characteristics of strongly-typed object-oriented languages, such as Java or C#, to improve safety, security, and modularity.

At the same time, like documents, mashups will constantly evolve as the users incorporate new sources of information or change the way it is processed. Unlike a Java program, a mashup will not reside at a single location; it will be a distributed data structure, and will share some of its content with other mashups. There will be no distinction between design time and runtime; a mashup will be collaboratively edited while it is being accessed. Finally, mashup elements will need to interact with one-another even if they share nothing in common except the data format; they will often come from arbitrary sources and share no common APIs.

A dedicated language for mashups will need to combine the characteristics of documents and programs. The LDO markup language already implements this idea to a limited degree [5, 9], and can serve as a basis for further research. I would like to evolve it into a general-purpose high-level programming environment.

In addition to the language, I would like to also focus on the client-side runtime environment. In many respects, LDO-style mashups go against the current trends; they inherently require P2P connectivity, cross-site scripting, and dynamically loaded code. Given such constraints, ensuring security is a major challenge.

References

[1] Ostrowski, K., Birman, K.: Extensible Web Services Architecture for Notification in Large-Scale Systems. In: ICWS '06: Proceedings of the IEEE International Conference on Web Services, Washington, DC, USA, IEEE Computer Society (2006) 383–392

[2] Ostrowski, K., Birman, K., Dolev, D.: Extensible Architecture for High-Performance, Scalable, Reliable Publish-Subscribe Eventing and Notification. International Journal of Web Services Research **4**(4) (2007) 18–58

[3] Ostrowski, K., Birman, K., Dolev, D.: Object-Oriented Architecture for Web Services Eventing. In Zhang, L.J., ed.: Web Services Research for Emerging Applications: Discoveries and Trends. IGI Global (2010)

[4] Ostrowski, K., Birman, K., Dolev, D.: Live Distributed Objects: Enabling the Active Web. IEEE Internet Computing **11**(6) (2007) 72–78

[5] Ostrowski, K., Birman, K.: WS-OBJECTS: Extending Service-Oriented Architecture with Hierarchical Composition of Client-Side Asynchronous Event-Processing Logic. In: ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services, Washington, DC, USA, IEEE Computer Society (2009) 25–34 (invited for publication in the International Journal of Web Services Research).

[6] Birman, K., Cantwell, J., Freedman, D., Huang, Q., Nikolov, P., Ostrowski, K.: Building Collaboration Applications that Mix Web Services Hosted Content with P2P Protocols. In: ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services, Washington, DC, USA, IEEE Computer Society (2009) 509–518

[7] Birman, K., Cantwell, J., Freedman, D., Huang, Q., Nikolov, P., Ostrowski, K.: Edge Mashups for Service-Oriented Collaboration. Computer **42**(5) (2009) 90–94

[8] Ostrowski, K., Birman, K.: Storing and Accessing Live Mashup Content in the Cloud. In: LADIS '09: Proceedings of the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware. (2009)

[9] Ostrowski, K., Birman, K., Dolev, D., Ahnn, J.: Programming with Live Distributed Objects. In: ECOOP '08: Proceedings of the 22nd European Conference on Object-Oriented Programming, Berlin, Heidelberg, Springer-Verlag (2008) 463–489

[10] Ostrowski, K., et al.: Live Distributed Objects. http://liveobjects.cs.cornell.edu, http://liveobjects.codeplex.com

[11] Ostrowski, K., Birman, K., Dolev, D., Sakoda, C.: Implementing Reliable Event Streams in Large Systems via Distributed Data Flows and Recursive Delegation. In: DEBS '09: Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems, New York, NY, USA, ACM (2009) 1–14

[12] Ostrowski, K., Birman, K., Dolev, D.: Distributed Data Flow Language for Multi-Party Protocols. In: PLOS '09: Proceedings of the 5th ACM SIGOPS Workshop on Programming Languages and Operating Systems. (2009)

[13] Ostrowski, K., Birman, K., Dolev, D.: Programming Live Distributed Objects with Distributed Data Flows. Cornell University Technical Report (2009) http://hdl.handle.net/1813/12766.

[14] Ostrowski, K., Birman, K.: Scalable Group Communication System for Scalable Trust. In: STC '06: Proceedings of the first ACM workshop on Scalable Trusted Computing, New York, NY, USA, ACM (2006) 3–6

[15] Ostrowski, K., Birman, K., Dolev, D.: QuickSilver Scalable Multicast (QSM). In: NCA '08: Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications, Washington, DC, USA, IEEE Computer Society (2008) 9–18

[16] Ostrowski, K., Sakoda, C., Birman, K.: Self-Replicating Objects for Multicore Platforms. Submitted to the 24th European Conference on Object-Oriented Programming (ECOOP 2010)

[17] Mundie, C.: Rethinking computing. Craig Mundie College Tour. Cornell University. (November 2009)