Programming Clouds

James Larus

Microsoft Research
One Microsoft Way
Redmond, WA 98052
larus@microsoft.com

Abstract. Cloud computing provides a platform for new software applications that run across a large collection of physically separate computers and free computation from the computer in front of a user. Distributed computing is not new, but the commodification of its hardware platform—along with ubiquitous networking; powerful mobile devices; and inexpensive, embeddable, networkable computers—heralds a revolution comparable to the PC.

Software development for the cloud offers many new (and some old challenges) that are central to research in programming models, languages, and tools. The language and tools community should embrace this new world as fertile source of new challenges and opportunities to advance the state of the art.

Keywords: cloud computing, programming languages, software tools, optimization, concurrency, parallelism, distributed systems.

1 Introduction

As I write this paper, cloud computing is a hot new trend in computing. By the time you read it, the bloom may be off this rose, and with a sense of disillusionment at yet another overhyped fad, popular enthusiasm may have moved on to the next great idea. Nevertheless, it is worth taking a close look at cloud computing, as it represents a fundamental break in software development that poses enormous challenges for the programming languages and tools.

Cloud computing extends far beyond the utility computing services offered by Amazon's AWS, Microsoft's Azure, or Google's AppEngine. These services provide a foundation for cloud computing by supplying on-demand, internet computing resources on a vast scale and at low cost. Far more significant, however, is the software model this hardware platform enables; one in which software applications are executed across a large collection of physically separate computers and computation is no longer limited to the computer in front of you. Distributed computing is not new, but the commodification of its hardware platform—along with ubiquitous networking; powerful mobile devices; and inexpensive, embeddable, networkable computers—may bring about a revolution comparable to the PC.

Programming the cloud is not easy. The underlying hardware platform of clusters of networked parallel computers is familiar, but not well supported by programming models, languages, or tools. In particular, concurrency, parallelism, distribution, and

R. Gupta (Ed.): CC 2010, LNCS 6011, pp. 1–9, 2010.
© Springer-Verlag Berlin Heidelberg 2010

availability are long-established research areas in which progress and consensus has been slow and painful. As cloud computing becomes prevalent, it is increasingly imperative to refine existing programming solutions and investigate new approaches to constructing robust, reliable software. The languages and tools community has a central role to play in the success of cloud computing.

Below is a brief and partial list of areas that could benefit from further research and development. The discussion is full of broad generalizations, so if I malign or ignore your favorite language or your research, excuse me in advance.

1. **Concurrency.** Cloud computing is an inherently concurrent and asynchronous computation, in which autonomous processes interact by exchanging messages. This architecture gives raise to two forms of concurrency within a process:

- The first, similar to an operating system, provides control flow to respond to inherently unordered events.
- The second, similar to a web server, supports processing of independent streams of requests.

Neither use of concurrency is well supported by programming models or languages. There is a long-standing debate between proponents of threads and event handling [1-3] as to which model best supports concurrency. Threads are close to a familiar, sequential programming model, but concurrency still necessitates synchronization to avoid unexpected state changes in the midst of an apparently sequential computation. Moreover, the high overhead of a thread and the cost of context switching limits concurrency and constrains system architectures. Event handlers, on the other hand, offer low overhead and feel more closely tied to the underlying events. However, handlers provide little program structure and scale poorly to large systems. They also require developers to explicitly manage program state. Other models, such as state machines or Actors, have not yet emerged in a general-purpose programming language.

2. **Parallelism.** Cloud computing runs on parallel computers, both on the client and server. Parallelism currently is the dominate approach to increasing processor performance without exceeding power dissipation limitations [4]. Future processors are likely to become more heterogeneous, as specialized functional units greatly increase performance or reduce power consumption for specific tasks.

Parallelism, unfortunately, is a long-standing challenge for computer science. Despite four decades of experience with parallel computers, we have not yet reached consensus on the underlying models and semantics or provided adequate programming languages and tools. For most developers, shared-memory parallel programs are still written in the assembly language of threads and explicit synchronization. Not surprisingly, parallel programming is difficult, slow, and error-prone and will be a major impediment in developing high-performance cloud applications.

The past few years have seen promising research on new, higher-level parallel programming models, such as transactional memory and deterministic execution [5, 6]. Neither is a panacea, but both abstractions could hide some complexities of parallelism.

3. **Message passing.** The alternative to shared-memory parallel programming is message passing, ubiquitous on the large clusters used in scientific and technical

computing. Because of its intrinsic advantages, message passing will be the primary parallel programming model for cloud computing as well. It scales across very large numbers of machines and is suited for distributed systems with long communications latencies. Equally important, message passing is a better programming model than shared memory as it provides inherent performance and correctness isolation with clearly identified points of interactions. Both aspects contribute to more secure and robust software systems [7].

Message passing can be more difficult to program than shared memory, in large measure because it is not directly supported by many programming languages. Message-passing libraries offer an inadequate interface between the asynchronous world of messages and the synchronous control flow of procedure calls and returns. A few languages, such as Erlang, integrate message into existing language constructions such as pattern matching [8], but full support for messages requires communications contracts, such as Sing# [9], and tighter integration with the type system and memory model.

4. **Distribution.** Distributed systems are a well-studied area with proven solutions for difficult problems such as replication, consistency, and quorum. This field has focused considerable effort on understanding the fundamental problems and in formulating efficient solutions. One challenge is integrating these techniques into a mainstream programming model. Should they reside in libraries, where developers need to invoke operations at appropriate points, or can they be better integrated into a language, so developers can state properties of their code and the run-time system can ensure correct execution?

5. **High availability.** The cloud end of cloud computing provides of services potentially used by millions of clients, and these services must be highly available. Failures of systems used by millions of people are noteworthy events widely reported by the media. And, as these services become integrated into the fabric of everyday life, they become part of the infrastructure that people depend on for their businesses, activities, and safety.

   High availability is not the same as high reliability, the focus of much research on detecting and eliminating software bugs. A reliable system that runs slowly under heavy load may fail to provide a necessary level of service. Conversely, components of a highly available system can fail frequently, but a properly architected system will continue to provide adequate levels of service [10].

   Availability starts at the architecture level of the system, but programming languages have an important role to play in the implementation. Existing language provide little support for systematically handling unexpected and erroneous conditions beyond exceptions, which are notoriously difficult to use properly [11]. Error handling is complex and delicate code that runs when program invariants are violated, but it is often written as an afterthought and rarely thoroughly tested. Better language support, for example lightweight, non-isolated transactions, could help developers handle and recover from errors [12].

6. **Performance.** Performance is primarily a system-level concern in cloud computing. Many performance problems involve shared resources running across large numbers of computers and complex networks. Few techniques exist to analyze a design or

system in advance, to understand bottlenecks or predict performance. As a consequence, current practice is to build, overprovision, measure, tweak, and pray.

One pervasive concern is detecting and understanding performance problems. Amazon's Dynamo system uses service-level agreements (SLA) among system components to quickly identify performance problems [13]. These SLAs are the performance equivalents of pre- and post-conditions. Making performance into a first-class programming abstraction, with full language and tools support, would help with the construction of complex, distributed systems.

7. **Application partitioning.** Current practice is to statically partition functionality between a client and service by defining an interface and writing both endpoints independently. This approach leads to inflexible architectures that forego opportunities to migrate computations to where they could run most efficiently. In particular, battery powered clients such as phones are limited in memory or processing capability. Migrating a running computation from a phone to a server might enable it to complete faster (or at all) or to better utilize limited network bandwidth by moving computation to data rather than the reverse [14].

   Even within a data center, code mobility is valuable. It permits server workloads to be balanced to improve performance or consolidated to reduce power consumption. Currently virtual machines move an entire image, from the operating system up, between computers. Finer-grain support for moving computations could lower the cost of migration and provide mechanisms useful in a wider range of circumstances.

   Statically partitioned systems could benefit from better language support. Microsoft's prototype Volta tool offered a single-source programming model for writing client-server applications [15]. The developer writes a single application, with annotations as to which methods run on the client or server. The Volta compiler partitions the program into two executables, a C# one for running on the server and a Javascript one for the client. Similar programming models could simplify the development of cloud applications by providing developers with a higher-level abstraction of their computation.

8. **Defect detection.** Software defect detection has made considerable progress over the past decade in finding low-level bugs in software. The tools resulting from this effort are valuable to cloud computing, but are far from sufficient. Few tools have looked for bugs in complex systems built from autonomous, asynchronous components. Although this domain appears similar to reactive systems, the complexity of cloud services present considerable challenges in applying techniques from this area.

9. **High-level abstractions.** Google's Map-Reduce and Microsoft Dryad are two higher level programming models that hide much of the complexity of writing a server-side analytic application [16, 17]. A simple programming model hides much of the complexity of data distribution, failure detection and notification, communication, and scheduling. It also opens opportunities for optimizations such as speculative execution. These two abstractions are intended for code that analyzes large amounts of data. There is a pressing need for similarly abstract models for writing distributed client-server applications and web services.

This list of open problems is not exhaustive, but instead is a starting point for research directly applicable to problems facing developers of cloud computing applications.

2 Orleans

Orleans is a project under development in the Cloud Computing Futures (CCF) group in Microsoft Research. Its goal is to achieve significant improvements in productivity of building cloud computing applications. Orleans specifically addresses the challenges of building, deploying, and operating very large cloud applications that encompass thousands of machines in multiple datacenters, constantly evolving software, and large teams to construct, maintain, and administer these properties.
At a coarse level, Orleans consists of three interdependent components:

- Programming model
- Programming language and tools
- Runtime.

Software for a cloud application, both the portion that runs on servers in a data center and the part that runs on clients, will be written in DC#, an extended version of C# that provides explicit support for the Orleans programming model. Orleans tools help a developer build reliable code by providing static and dynamic defect detection and test tools. Application code runs on the Orleans run-time system, which provides robust, tested implementations of the abstractions needed for these systems. These abstractions in turn execute on Azure, Microsoft's data center operating system.

2.1 Design Philosophy

Orleans is frankly a prescriptive system—it strongly encourages the use of software architectures and design patterns that have proven themselves in practice. Because Orleans targets large-scale cloud computing, the key criterion for adopting a principle is that it results in a scalable, resilient, reliable system. Cloud software is scalable if it a system can grow to accommodate a steadily increasing number of clients without requiring major rewrites, even when the increase in volume spans multiple orders of magnitude. The common practice today is to plan on several complete rewrites of a system as an internet property grows in popularity, even though there are multiple examples of scalable internet properties whose design principles are widely known. Today's general-purpose programming languages and tools provide little or no support for these principles, so the burden of scalability is shifted to developers; and consequently most new enterprises choose short-term expediency to get their websites up quickly.

A system is resilient if it can tolerate failures in its components: the computers, communication network, other services on which it relies, and even the data center in which it runs. Toleration requires the system to detect a failure, respond to it in a manner that minimizes the effect of a failure on unrelated components and clients, restore service when possible by using other resources, and resume execution when the failure is corrected.

The distributed systems community has studied techniques for building scalable, resilient software systems for many years. A small number of abstractions have proven their value in building these systems: asynchronous communications and software architecture; data partitioning; data replication; consensus; and consistent, systematic design policies. Orleans will build these ideas into its programming and data model and provide first-class support for them in the DC# language and tools. These abstractions by no means guarantee a well-written program or successful system; it still remains true that it is possible to write a bad program in any language. However, these abstractions have proven their value in many systems and are well studied and understood, and they provide a solid basis for building resilient systems.

2.2 Centrality of Failure

In ordinary software, error-handling code is home to a disproportionate share of defects. This code is difficult to write because invariants and preconditions often are invalid after an error and paths through this code are less well tested because they are uncommon. Distributed systems complicate error handling by introducing new failure modes, such as asynchronous communications and partial failure, which are challenging to reason about and difficult to handle correctly. Much of the difficulty of building a reliable internet property is attributable to asynchrony and failure.

Distributed systems research offer some techniques for masking failures and asynchrony (e.g., Paxos), but they have significant drawbacks and are unsuitable to mask all failures in a responsive service. Paxos and other replication strategies increase the quantity of resources dedicated to a computation task by a significant (3 – 5x) amount. In addition, these techniques increase the time to perform an operation. Because of increased cost and latency, replication strategies must be used sparingly in scalable services.

Other techniques, such as checkpoint and restart, are more successful for non-reactive computations (e.g., large-scale analytic computations implemented with map-reduce or Dryad) in which it is possible to capture input to a portion of a computation and in which a large recovery cost is less than the far-more-expensive alternative of rerunning the entire computation. Another advantage is that it is possible to automate the failure detection and error recovery process.

Programming models also have a significant influence on the correctness and resiliency of code. For example, every client making a remote procedure call (RPC) has to deal with three possibilities: the call succeeds and the client knows it; the call fails and the client knows it; the call times out and the client does not know whether it succeeded or failed. In more sophisticated models that allow simultaneous RPC calls, complexity further increases when calls complete in arbitrary orders. Complicating this reasoning is the syntactic similarity of an RPC call and a conventional call, which encourage a developer to conflate the two, despite their vast difference in cost and semantics. For these reasons, undisciplined use of RPC has proven to be a bad abstraction for building distributed systems.

2.3 Orleans Programming Model

The Orleans programming model is inherently more resilient. An application is composed of loosely coupled components, each of which executes in its own failure

container. In Orleans, these components are called grains. A grain consists of a single-threaded computation with its local state. It can fail and be restarted without directly affecting the execution of any other grain—though it may indirectly affect a dependent grain that cannot respond appropriately to its failure. All communications between grains occurs across channels: higher-order (i.e., can send a channel over a channel), strongly typed paths for sending messages between grains. The code within a grain is inherently asynchronous, to deal with the unpredictable arrival of messages across multiple channels or the unpredictable ordering of messages between asynchronous services. This model exposes the reality of a distributed system (communication via messages that arrive at unpredictable times) but constrains it, in single threaded, isolated containers, to simplify reasoning about and analyzing code.

Grains are not distributed objects. The differences between the two models are fundamental. Orleans does not provide a pointer or reference to a grain, nor do grains reside in a global address space. A computation communicates with a grain through a channel, which is a capability, not a reference. A common channel allows two grains to communicate according to the channel's protocol. However, the channel does not uniquely identify either grain since channels can be passed around. Nor does a channel identify the location of a grain, which can migrate between machines while the channel is active.

Moreover, interactions between grains are asynchronous, not RPC. One grain can request another grain perform an operation by sending a message (which could be wrapped in syntactic sugar to look like a method invocation). The receiving grain has the freedom to process this request in any order with respect to its on-going computations and other requests. When the operation completes, the grain can send back its result. In general, the first grain will not block waiting for this value, as it would for a method call, but instead will process other, concurrent operations.

An important property of a grain is that it can migrate between computers. Migration allows Orleans to adaptively execute a system: to reduce communication latency by moving a computation closer to a client or data resource, to increase fault tolerance by moving a computation to a less tightly coupled system, and to balance the load among servers.

Grains encourage an SPMD (single program, multiple data) style of programming. The same computation (code) runs in all grains of a particular type, and each grain's computation executes independently of other grains and the computations are initiated at different times.

However, it is also possible to use grains to implement a dataflow programming model. In this case, a grain is a unit of computation that accepts input and sends results across channels. Dataflow is appropriate for streaming computation and can achieve the scalability of asynchronous data parallelism by replicating dataflow graphs and computations.

What is the appropriate size for a grain? In today's scalable services, it is necessary to partition the data manipulated by the service at a fine granularity, to allow for rebalancing in the face of load and usage skew. For example, code acting on behalf of a Messenger user does not assume it is co-located with another Messenger user, and it must expect the location of a user's data to change when a server is added or removed. Similar properties hold for Hotmail user's address books, subscriptions in Live Mesh's pub-sub service, ongoing meetings in Office Communications Server, rows in Google's BigTable, keys in Amazon's Dynamo, etc. With this fundamental

assumption, a system can spread a large and varying collection of data items (e.g., a user's IM presence) across a large number of servers, even across multiple data centers. Though partitioning by user is illustrative, grains can represent many other entities. For example, a user's mailbox may contain grains corresponding to mail messages.

2.4 Orleans Data Model

Data in cloud computing application exists in a richer, more complex environment than in non-distributed applications. This environment has a number of orthogonal dimensions. Unlike the local case, a single model does not meet all needs. Different grains will require different guarantees, and the developer must assume responsibility for selecting the properties that match the importance of data, semantics of operations, and performance constraints on the system. Orleans will implement a variety of different types of gains that support the different models for the data they contain, so an application developer can declare the properties of a particular grain and expect the system to implement its functionality.

Data can be persistent, permitting it to survive a machine crash. Changes to the data are written to durable storage and Orleans can keep independent copies of the data on distinct computers (or data centers), to increase availability in the face of resource failures.

Replicating the data among machines introduces the issue of consistency among the replicas. Strong consistency requires the replicas to change simultaneously, while weaker models tolerate divergence among the copies.

Within a grain, Orleans supports a simple, local concurrency model. Data local to the grain is only modified by code executing in the grain and execution is single-threaded, so from the perspective of this code, the execution model is mostly sequential. However, when code for an operation ends and yields control back to the grain, other operations can execute and modify the grain's local state, so a developer cannot make assumptions across turns in a grain.

Orleans does not impose a single model on the operations exported by a grain. The semantics of concurrent operations has been formalized in numerous ways, and different models (e.g., sequential consistency, serializability, linearizability) offer varying tradeoffs among simplicity, generality, and efficiency. Orleans will need to provide the support that enables a developer to implement these models where appropriate.

3 Conclusion

Until recently, only a handful of people had ever used more than one computer to solve a problem. This is no longer true, as search engines routinely execute a query across a thousand or so computers. Cloud computing is the next step into a world in which computation and data are no longer tightly tied to a specific computer and it is possible to share vast computing resources and data sets to build new forms of computing that go far beyond the familiar desktop or laptop PCs.

Software development for the cloud offers many new (and some old challenges) that are central to research in programming models, languages, and tools. The language and tools community should embrace this new world as fertile source of new challenges and opportunities to advance the state of the art.

References

1. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative Task Management without Manual Stack Management or, Event-driven Programming is Not the Opposite of Threaded Programming. In: Proceedings of the USENIX 2002 Conference, pp. 289–302. Usenix, Monterey (2002)
2. Ousterhout, J.: Why Threads are a Bad Idea (for most purposes). In: Proceedings of the 1996 USENIX Technical Conference. Usenix, San Diego (1996)
3. von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: Scalable Threads for Internet Services. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 268–281. ACM, Bolton Landing (2003)
4. Larus, J.: Spending Moore's Dividend. Communications of the ACM 52, 62–69 (2009)
5. Larus, J., Kozyrakis, C.: Transactional Memory. Communications of the ACM 51, 80–88 (2008)
6. Bocchino Jr., R.L., Adve, V.S., Adve, S.V., Snir, M.: Parallel Programming Must Be Deterministic by Default. In: First USENIX Workshop on Hot Topics in Parallelism. Usenix, Berkeley (2009)
7. Hunt, G., Larus, J.: Singularity: Rethinking the Software Stack. ACM SIGOPS Operating Systems Review 41, 37–49 (2007)
8. Armstrong, J.: Programming Erlang: Software for a Concurrent World. The Pragmatic Bookshelf, Raleigh (2007)
9. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language Support for Fast and Reliable Message Based Communication in Singularity OS. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, Leuven, Belgium, pp. 177–190 (2006)
10. Barroso, L.A., Hölzle, U.: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, vol. 6. Morgan & Claypool, San Francisco (2009)
11. Weimer, W., Necula, G.C.: Exceptional Situations and Program Reliability. ACM Transactions on Programming Languages and Systems 30, 1–51 (2008)
12. Lenharth, A., Adve, V.S., King, S.T.: Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 49–60. ACM, Washington (2009)
13. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. ACM, Stevenson (2007)
14. Gray, J.: Distributed Computing Economics. Microsoft Research, p. 6. Redmond, WA (2003)
15. anon.: Volta Technology Preview from Microsoft Live Labs Helps Developers Build Innovative, Multi-Tiered Web Applications with Existing Tools, Technology. Microsoft Press Pass (2007)
16. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 51, 107–113 (2008)
17. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pp. 59–72. ACM, Lisbon (2007)