# Interaction-Based Programming Towards Translucent Clouds (Position Paper)

Yu David Liu and Kartik Gopalan
SUNY Binghamton
{davidL, kartik}@cs.binghamton.edu

## Abstract

Today's cloud computing platforms are typically "opaque": Amazon EC2 users only receive *virtual* units of CPU and memory, and *physical* details of the platform are hidden. Such opacity prevents programs from online optimizations and deployment adjustment, and is penalizing the very applications cloud computing attempts to attract: high-performance software. On the other extreme, a completely transparent design of clouds would lead to severe security and reliability concerns. In this position paper, we take the middle-of-the-road approach, proposing a language model for well-defined programmable interactions between the cloud platform and the client program. Our proposed ideas draw from the previous work of Classages, where first-class interactions can be dynamically established and terminated for two parties with mutually satisfiable contracts. In this paper, we justify how our design can help cloud programs to fine-tune performance, and how it may impact on other important issues of cloud computing, such as security, scheduling, and pricing.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Organization and Design—Distributed Systems

***General Terms*** Languages

***Keywords*** Cloud programming, interactions

## 1. Introduction

Cloud computing provides an attractive technology for end users to "lease" powerful hardware platforms remotely, and pay bills as if they were paying for electricity and gas. Most major cloud service providers today tend to prefer the "opaque" cloud model: users submit processing and data storage requests to the cloud without fully knowing where, when, and how their requests are handled. For example, Amazon EC2 instances, Google App Engine quota resources, Microsoft Azure compute instances, and Rackspace Cloud servers are all *virtual* units of CPUs, memory and storage. These units are more of quality of service (QoS) guarantees than physical details of the cloud infrastructure itself. Indeed, the very terminological origin of *cloud* computing alludes to the opaque nature of the cloud platforms.

There are strong reasons why cloud platforms should not be completely transparent to end users. Cloud service providers are in business to make a profit. This goal translates into two operational requirements: (1) satisfying the requirements of cloud applications by providing greater QoS, and (2) minimizing the resource usage in their cloud platforms, in terms of hardware infrastructure, power, and management costs. These two goals are often in conflict with each other; to achieve a balance, the cloud service providers are increasingly adopting virtualization technology [VMW, BDF+03, KKL+07] – the ability to transparently execute multiple virtual machines on a single physical machine. Virtualization enables the service providers to boost resource utilization by packing more work into fewer physical machines. Virtualization also enables flexibility in resource administration through the use of live migration [CFH+05, NLH05, HDG09].

However, higher resource utilization and administration flexibility comes at the cost of application-specific performance optimizations. The cloud application writers do not know when, where and how their applications would execute. Greater visibility into the internals of the cloud platforms could enable application writers to better optimize their applications. For example, from a scheduling perspective, a web service application may prefer being co-scheduled (scheduled simultaneously) with its back-end database server in order to minimize its communication latency. Similarly, a geographically distributed content delivery system may require that its servers be placed at different physical locations at various times of the day to better optimize its interactions with users. As yet another example, distributed high performance computing applications might require maximum bounds on communication latency between any two nodes involved in collaborative computation activity. Such usage instances call for cloud providers to en-



*APLWACA '10* June 6, Toronto, Canada.

able cloud applications to be aware of the details of their execution environment.

In this position paper, we propose a middle-of-the-road approach between complete transparency and complete opaqueness, which we term *translucent cloud programming*. It is our belief that controlled exposure of the cloud infrastructure to application writers – and consequently principled interaction between the cloud and the applications on it – can only help satisfy the two previous operational requirements, beneficial to both cloud users and cloud providers. For cloud users, a greater awareness of platform details may lead to application-specific optimizations. For cloud providers, further resource usage reductions may be achieved with a greater awareness of the application status. For the second part, imagine cloud software is developed in a language where the program run-time carries information on *how* "active" a computation is. The cloud can then make scheduling, paging, and caching decisions accordingly, *e.g.* preferring those more "active" computations.

Concretely, we propose a new object-oriented programming language, *iCloud*, for cloud computing. *iCloud* is designed with the philosophy of Interaction-Based Programming: interactions should be first-class citizens of programs, and should only happen on explicitly defined interfaces. Interaction-Based Programming is also the central theme of Classages [LS05], a general-purpose language for representing first-class object relationships. In contrast with its precursor, the design of *iCloud* is more focused on how the dynamic aspects of interactions can help address important issues in cloud programming, which we elaborate in the rest of the paper.

## 2. Translucent Cloud Programming

We demonstrate the key ideas of our language with an example, in Fig. 1 and Fig. 2. The oversimplified cloud application here is composed of two concurrent parts, main program logic in `MyApp` and database-related operations in `MyDB`. A third class, `CloudCPUServices` in Fig. 2, is the cloud provider's abstraction for the scheduling service. With this class, the application can interact with the cloud directly to perform scheduling-related optimizations – the cloud is thus no longer completely opaque from the programmer's perspective.

***Connectors for Controlled Cloud Service Exposure*** Similar to Java, an *iCloud* class may also have local fields and local methods; what distinguishes it from a Java one is its *connectors* – the interfaces defining what interactions its run-time instances can be involved in. For instance, class `MyApp` has a `DB` connector and `Sch` connector. This means that a `MyApp` object can participate in *two and only two* kinds of interactions with other objects at run time.

What is relevant to our discussion here is cloud services can also be abstracted as classes with connectors. *iCloud* does not allow public field or method access, so connec-

```
class MyApp {
    MyApp(CloudCPUServices ccs, MyDB db) {
        csh = connect ccs with Sch  >> ISch;
        db = db;
    }
    connector DB {
        import query(Query);
        export queryDone(DataSet d) {
            ...process result d ...
            disconnect;
        }
    }
    connector Sch {
        import coschedule(Object);
        import setTID(Object);
        export coscheduleDone() {}
        export scheduleFriends() { }
    }
    void process(Query q) {
        cdb = connect db with DB  >> Data;
        cdb -> setTID(this);
        cdb -> query(q);
        csh -> coschedule(db);
    }
    Sch csh;
    DB db;
    ...other local fields and methods ...
}


class MyDB {
    MyApp(CloudCPUServices ccs) {
        connect ccs with Sch  >> ISch;
    }
    connector Data {
        export void query(Query q) {
            DataSet d= ... prepare result ...
            queryDone(d);
        }
    }
    connector Sch {
        import coschedule(Object);
        export scheduleFriends() {
            for(i =0; i< FRIENDNUM; i++)
                coschedule(friends[i]);
        }
    Object friends[FRIENDNUM];
}
```

**Figure 1.** A Cloud Application

```
class CloudCPUServices {
    connector ISch {
        state Object clientTID;
        export coschedule(Object friendID) {
            a1 = affinity clientTID;
            a2 = affinity friendID;
            if distance(a1, a2) > THRES {
                a3 = findAvailable(a1);
                migrate friendID to a3;
                forall(c: ISch)
                    if(c -> clientTID == friendID)
                        c -> scheduleFriends();
            }
            coscheduleDone();
        }
        export setTID(Object tID) {
            clientTID = tID;
        }
        import  scheduleFriends();
        import coscheduleDone();
    }
    connector ICache {
        ...
    }
    int findAvailable(int center, int dist) {
        //return a CPU close to center within distance dist
    }
    int distance(int a1, int a2) {
        //return the "distance" of CPUs a1 and a2
    }
}
```

**Figure 2.** A Cloud Service

tors serve as the complete specification of the cloud's exposure to the programmer. To avoid "too much" transparency, cloud service providers need to, and only need to, design connectors carefully. For example, the ISch connector of CloudCPUServices exposes a method called coschedule, which allows whoever interacts with this connector to coschedule threads.

A connector-based design is a boon for security, so that access control policies are only needed on these well-defined interfaces [LS02, LS06]. Information-flow related verification may also become simpler, because it is the data that cross the boundary of connectors are what really matter.

Each connector may have a number of import's and export's. Each export is a method the connector can provide to "the other party" (*i.e.* whoever is connected to this connector) , and each import is a signature specifying what it expects the other party to provide. Such bi-directional dependencies express the "mutual satisfaction" nature of interactions. Each connector may also hold connection-specific data. For example, connector ISch of CloudCPUServices contains a mutable state clientTID, which keeps track of the thread ID of the party concurrently connected to ISch.

***Connections have a Dynamic Lifespan*** Constructs such as import and export are commonly designed in module systems and linking calculi [BC90, Car97, FKF98, FF98]. Interactions on connectors however are not linking. Such interactions, called *connections*, can instead be viewed as dynamically established and terminated pathways for message passing. To establish a connection between the Sch connector of a MyApp object and the ISch connector of a CloudCPUServices object (say s), the MyApp object at runtime can evaluate the following expression:

```
c = connect s with Sch >> ISch;
```

The result of the connect expression, c, is called a *connection handle*, the first-class incarnation of the message pathway. After the connection is established, it stays alive, so that message coschedule can be passed along from the MyApp object to the CloudCPUServices object, via expression c->coschedule(3). The connection is terminated via expression disconnect c, denoting connection c is disconnected. If the expression appears inside a connector, the argument can be ignored, denoting the current connection is disconnected.

Note that connections are established asymmetrically: connect is always evaluated in the scope of the connection initiator. For the party being connected, the object can always query all the connection handles for a particular connector, via forall syntax. For instance in CloudCPUServices, the forall(c : ISch){...} expression enumerates all live connections currently associated with connector ISch.

In the context of cloud computing, dynamic lifespan of connections can be very useful for process and memory management – each use of disconnect in the program expresses the application programmer's intention of "I won't be using this interaction for a while," valuable information for the cloud platform to decide how to utilize precious CPU and memory resources. New scheduling and paging algorithms can be designed to count the live connections associated with each connector periodically, and those objects with no live connections can be paged out of memory, and their operating threads can be context-switched out. Observe that the strategy here is very different from application-blind algorithms, such as garbage collection: an object can very well still be referenced but there are no connections to it at a specific point in time.

***Pricing Model*** With different kinds of services encapsulated into distinct connectors, the use of each connector can be priced separately to meet varied functionality and QoS requirements. Advanced cloud service connectors may only be accessed by users willing to pay an extra charge. Cloud providers have the financial incentive for this pricing strategy, and cloud users are satisfied for better services.

Connections with a dynamic lifespan may also lead to new pricing strategies for cloud computing. Users can be charged only for the duration when each connection is alive.

On today's cloud computing platforms, the most common strategy is to charge users based on the time the application is running. This is suboptimal for several reasons. First, it does not provide incentives for users to disconnect as often as they can, which in turn would prevent smarter resource optimization and management on the end of cloud providers as we described earlier. Second, users are in general given no incentive to save as much resource and use as little service as they can, which eventually leads to the overall waste of power and computing resources.

Other popular pricing strategies applied by today's commercial cloud services include those based on bandwidth, storage, or system load (auction-based). These more QoS-oriented metrics are certainly an improvement over the time-based flat-rate plans, but none of these truly link the unit of pricing with that of the program itself. With *iCloud*, service providers are encouraged to offer more diversified paid services, encapsulating each with a connector with its own pricing strategy; In the meantime, cloud application writers can program with pricing implications in mind – the choice of connectors and the duration of keeping each connection alive both reflect the programmer's desire on how to balance QoS and cost.

***Concurrency Model*** Cloud computing is fundamentally associated with parallel computing. In *iCloud*, a concurrency model similar to Actors [Agh90] is adopted: each object lives in its own thread and can be deployed on different parallel units of the cloud. With the uniform treatment of objects and threads, the object ID can double as a thread ID. For example, the argument `friendID` of `coschedule` is an object reference, but it can be used directly by the `CloudCPUServices` as a thread identifier for scheduling.

From this perspective, a connection is a message pathway between parallel threads. Following Actors and numerous non-shared memory message passing designs, message passing through a connection in *iCloud* is asynchronous. Methods in a connector do not have return values. Such a design does not present any expressiveness loss in an interface with both import and export however. In the `DB` connector, when the `query` message is sent, the execution is non-blocking – no result of the query is returned immediately; later when the `DB` thread is ready with the query result, the `queryDone` message will be sent back to `MyApp`. The Hollywood principle here ("I will call you back") should be familiar to most event-programming programmers. An identical design has appeared in sensor network programming [GLvB+03].

As an alternative solution, it is also possible to define some connectors as "intra-node" and others as "inter-node." With this slightly more elaborate design, messages through "intra-node" connections can be synchronous, a design that may appear more friendly to Java programmers. A language that allows for mixed asynchronous messaging and synchronous messaging is Coqa [LLS08].

***Cloud-Specific Primitives*** We use expression affinity $e$ to retrieve the CPU affinity associated with each thread $e$ (*i.e.* object ID). As scheduling becomes a central issue for cloud management, we introduce expression migrate $e$ to $e'$ to express thread $e$ is going to migrate to CPU $e'$.

***The Example Revisited*** The example in Fig. 1 and Fig. 2 is an oversimplified illustration of a widely known co-scheduling strategy [ADCM98]: whenever a potentially bandwidth-heavy request happens, co-schedule the receiver thread to be as close to the sender as possible. In our example, let us assume the database query will return a large `DataSet`. In method `process`, the scheduler is requested to co-schedule the `DB` object as close to `MyApp` as possible after the `query` is sent. The import of `coschedule` on the end of `MpApp` is implemented by the export on the end of `CloudCPUServices`. In that method, the "distance" of between `MyApp`'s CPU and `MyDB`'s CPU is computed, and necessary steps are taken to bring them closer. If `MyDB` has migrated from one CPU to another, the method checks whether any other threads in close interaction with `MyDB` (*i.e.* "friends") need to be coscheduled too.

## 3. Conclusion

This paper explores programming language support for cloud computing. The proposed *iCloud* language is aimed at providing principled programmable interactions between cloud services and cloud applications, and between different parts inside applications. The design of *iCloud* touches upon a number of important issues in cloud computing, such as concurrency, pricing, and security.

On the philosophical level, Translucent Cloud Programming favors "dialog, not confrontation" between cloud service providers and application writers. We believe the benefit is mutual: more knowledge of cloud physical details may help programmers perform application-specific optimizations such as scheduling and caching, whereas more knowledge of applications may help cloud infrastructure gain insight on the intentions and status of the applications and optimize accordingly.

Just as most new programming models, the programming idioms behind *iCloud* can either be implemented as a new language or a library to an existing language. We have chosen the first route here, because compiler-directed checking offers stronger guarantees to correctness; a language-based approach also enables more powerful static and dynamic optimizations. For instance, connector encapsulation and connector matching at interaction time can be formulated as type invariants and guaranteed by a sound static type system. A similar type system has been constructed for Classages, so we are confident it can be successfully constructed for *iCloud* too. As another example, the online optimization based on live connection counting we described in Sec.2 can be performed for *all* programs. In cases where new languages are not possible (*e.g.* legacy code upgrade), we are

open to the idea of implementing the idioms of *iCloud* as a library. A previous library-based implementation of connectors was explored by ArchJava [ASCN03].

Our language design is motivated by providing better interactions between cloud infrastructure and cloud applications. A future direction we would like to explore is to use the interaction-based model to program interactions between different parts of a cloud application. Programming models with strikingly different paradigms exist in this latter category, such as MapReduce [DG04] and BOOM [ACC+10]. It will be interesting to see how *iCloud* can blend with these existing models, and what additional benefits can be obtained.

## References


[ACC+10] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 223–236, 2010.

[ADCM98] Andrea C. Arpaci-Dusseau, David E. Culler, and Alan M. Mainwaring. Scheduling with implicit information in distributed systems. *SIGMETRICS Perform. Eval. Rev.*, 26(1):233–243, 1998.

[Agh90] Gul Agha. *ACTORS : A model of Concurrent computations in Distributed Systems*. MITP, Cambridge, Mass., 1990.

[ASCN03] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In *ECOOP'03*, pages 74–102, 2003.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the Joint ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and the 4th European Conference of Object-Oriented Programming (ECOOP)*, pages 303–311, 1990.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton, NY, USA*, pages 164–177, 2003.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 266–277, 1997.

[CFH+05] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of Network System Design and Implementation*, 2005.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*, 2004.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, 1998.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 171–183, 1998.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03*, pages 1–11, New York, NY, USA, 2003. ACM.

[HDG09] Michael Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *SIGOPS Operating Systems Review*, 43(3):14–26, July 2009.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proc. of the Linux Symposium*, pages 225–230, June 2007.

[LLS08] Yu David Liu, Xiaoqi Lu, and Scott F. Smith. Coqa: Concurrent objects with quantized atomicity. In *CC'08*, March 2008.

[LS02] Yu David Liu and Scott F. Smith. A Component Security Infrastructure. In *Foundation of Computer Security Workshop*, 2002.

[LS05] Yu David Liu and Scott F. Smith. Interaction-based Programming with Classages. In *OOPSLA'05*, pages 191–209, 2005.

[LS06] Xiaoqi Lu and Scott F. Smith. A microkernel virtual machine: building security with clear interfaces. In *PLAS*, pages 47–56, 2006.

[NLH05] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *In Usenix Annual Technical Conference*, 2005.

[VMW] VMWare Corporation. *VMWare ESX Server, available online at http://www.vmware.com/products/vi/esx/.*