Vision: a Lightweight Computing Model for Fine-Grained Cloud Computing

Justin Mazzola Paluska
MIT CSAIL
Cambridge, MA, U.S.A.
jmp@mit.edu

Hubert Pham
MIT CSAIL
Cambridge, MA, U.S.A.
hubert@mit.edu

Gregor Schiele
Universität Mannheim
Mannheim, Germany
gregor.schiele
@uni-mannheim.de

Christian Becker
Universität Mannheim
Mannheim, Germany
christian.becker
@uni-mannheim.de

Steve Ward
MIT CSAIL
Cambridge, MA, U.S.A.
ward@mit.edu

ABSTRACT

Cloud systems differ fundamentally in how they offer and charge for resources. While some systems provide a generic programming abstraction at coarse granularity, e.g., a virtual machine rented by the hour, others offer specialized abstractions with fine-grained accounting on a per-request basis. In this paper, we explore Tasklets, an abstraction for instances of short-duration, generic computations that migrate from a host requiring computation to hosts that are willing to provide computation. Tasklets enable fine-grained accounting of resource usage, enabling us to build infrastructure that supports trading computing resources according to various economic models. This computation model is especially attractive in settings where mobile devices can utilize resources in the cloud to mitigate local resource constraints.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Cloud Computing*

Keywords

Cloud Computing, Cloud Economy, Mobile Code Offload

1. INTRODUCTION

Mobile cloud computing is a promising trend in mobile systems. If an application requires more resources than what is available on a user's mobile device, the application offloads computation to a cloud service running either in a private, company-owned data center or a public, for-rent data center.

While some mobile applications, such as Apple's Siri speech recognizer, reside in a private data center, most applications rely on publicly-available data centers. Currently, there are two popular models for selling and accounting for use of cloud computing resources: (1) charging per usage hour for generic virtual machines, e.g., as in Amazon EC2 and Windows Azure, and (2) charging for more fine-grained service requests, e.g., as in Google App Engine. While the first approach of virtual machines offers maximum flexibility with respect to the programming model, their coarse-grained accounting restricts the kinds of jobs that may run. Because virtual machines require significant initial setup, they are best suited for large numbers of identical tasks that (1) may be combined to amortize machine setup and maintenance, and (2) may consume every rented minute of computation, as unused capacity goes to waste. In contrast, the fine-grained accounting model of Google App Engine enables small tasks to take up only small amounts of rented time, but requires the programmer to structure her application around the particular fine-grained request/response interfaces of App Engine. Figure 1 illustrates the spectrum of flexibility and the price.

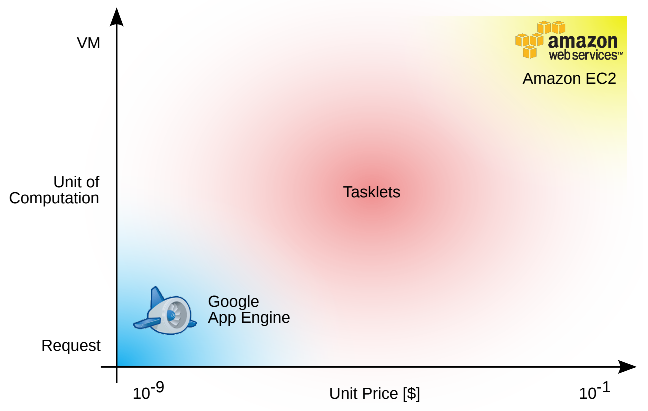

Figure 1: Spectrum of cloud resources as a function of computation unit size and unit price.

In this paper, we argue that mobile cloud systems need a programming model that offers the best of both worlds: fine-grained accounting with a flexible execution environment. Such a model enables mobile applications to offload even small amounts of computation to the cloud, or even to each other, with little overhead. The core enabler of our programming model is the *Tasklet*. A Tasklet is an abstraction of a particular thread of computation, encompassing both the computation and any data that the computation requires. Tasklets are supported by a Tasklet containers that may execute Tasklets locally as well as offload Tasklets to containers on other hosts. Tasklets are composed of chunks, a uniform abstraction for computation, data storage, and communication. Resource accounting is done by counting chunks—the number of chunks executed, the number of chunks stored, or the number of chunks transmitted across a network—as they are managed by a Tasklet container. Tasklets enable generic computation at fine-grained scales, thus filling the design space in the middle of Figure 1.

Tasklet containers may run as new cloud services (e.g., Tasklets as a Service), in idle time on top of existing cloud services (e.g., as a process that takes up the remaining slack time on an already running EC2 instance), or in idle time on locally available computing hardware (e.g., as in resource foraging systems). Tasklets, Tasklet containers, and the chunk-based accounting system enable a generic economy of computational resources. We argue that this will reduce the amount of wasted resources in a cloud system and thus enhance the system's efficiency while at the same time enabling offload of ad-hoc tasks.

This paper proposes an overall architecture where Tasklets serve as the abstraction of cloud computing. The Tasklet programming model is explained in the next section. Some possible economic models based on Tasklets are presented thereafter. Section 4 briefly discusses related work. The paper closes with a conclusion and outlook to future work.

2. TASKLETS

A Tasklet is a lightweight abstraction for a thread of computation that typically represents a single task, such as applying a filter to a photo or running speech recognition on a single utterance. We target such tasks because they can benefit from the additional power of the data center but would both (1) take too long to run within a single request to, for example, Google App Engine and (2) be too short to start a virtual machine and waste an hour of computation. Tasklets encapsulate both code and data so that a single Tasklet is a self-contained computation.

2.1 Chunks

Tasklets are composed of chunks [1], an abstraction we developed for exchanging data and computation in heterogeneous distributed systems. Chunks are fixed-sized arrays of fixed-sized *slots*. Slots may contain either scalar data or a link to another chunk. Link slots contain location-independent references to other chunks, allowing chunks to migrate between hosts as necessary. The primary advantage of chunks is that they encourage fine-grained decomposition of computations and data structures: since chunks are of a fixed-size, objects that overflow one chunk must be decomposed into a network of many chunks. This property enables efficient migration of chunks between Tasklet containers as well as precise accounting of resource usage.

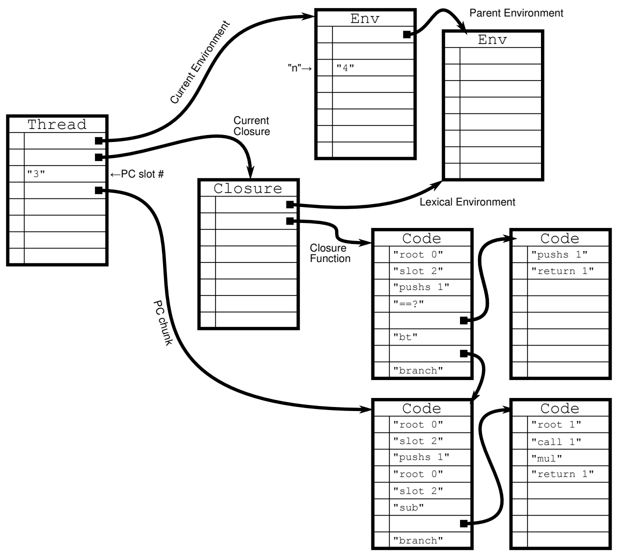

Figure 2: Example Tasklet computing factorial.

Tasklets represent data, code, and machine runtime state in chunks, as shown in Figure 2. Conceptually, a Tasklet realizes a closure and thus contains the function and the data required for computation. The core of a Tasklet is a Thread chunk. Threads contain a program counter and references to the program and the environment. The program consists of blocks of byte code whose links represent targets of control structures, e.g., loops or conditional statements. Program data is linked from the environment, enabling code to operate on it and return execution results by modifying the environment. The Tasklet virtual machine uses a stack-based, chunk-oriented byte code to express operations. The example in Figure 2 depicts a simple computation of factorial of n, with computation paused at $n = 4$. In practice, programmers develop in JavaScript or other languages and rely on chunk-aware compilers to generate byte code.

It is possible to migrate a computation between Tasklet containers by pausing the computation, shipping the Tasklet's chunks to the remote container, and resuming computation with the transfered chunks. In contrast to an EC2 virtual machine that needs to be appropriately configured before use, all Tasklet containers are "blank slates" that rely on a Tasklet linking to all of the data and libraries that it needs in order to execute.

2.2 Tasklet Accounting with Chunks

Proper accounting for resource usage is crucial in building an economy to utilize available capacity. The fixed-size nature of chunks enables our Tasklet layer to account for resource usage by counting chunks. For example, bandwidth consumption is measured as the number of chunks transferred between two hosts while computation consumption is measured as the number of chunks executed. While it is possible to account for resource consumption using multiple different metrics, e.g., bytes for network transfers and milliseconds for compute time, using a single unit to measure resource usage makes it easier to build an economy (c.f. Section 3) since there is only one currency to trade.

The chunk-based foundation for Tasklets encourages fine-grained accounting by encouraging the use of (1) fine-grained and (2) single-purpose chunk structures. The fixed-sized nature of chunks forces the former since any structure larger than a single chunk must be broken apart into multiple chunks. To achieve the latter, the Tasklet byte code only allows branches to the beginning of chunks, forcing each basic block into its own chunk. For example, the recursive factorial function of Figure 2 decomposes into four chunks. With such a restriction, each chunk represents one, and only one, part of a computation. In the figure, factorial decomposes into three groups of chunks: one chunk for the base case, two chunks for the recursive case, and one chunk for the conditional code that decides which case is currently being computed. Similarly, the fined-grained nature of Tasklets enables us to make use of large libraries (e.g, for image filtering or speech recognition) or large data sets, but only transfer the specific parts of the libraries or data sets necessary to complete the Tasklet computation.

While recording runtime usage is important for simple accounting, a mobile client may wish to predict costs before making a decision to offload a Tasklet to a particular container. We envision that either static analysis or Tasklet execution history can help predict how much computing time is required to compute a Tasklet. For example, for the code in Figure 2, a static analyzer can determine an exact solution for the number of chunks executed. factorial(n) executes two chunks for the base case of $n = 1$ and three chunks for $n > 1$, leading to a closed form execution time of $3(n-1)+2$ chunks. For computations where analysis is more difficult, Tasklets offer graceful recovery for inaccurate analysis. For example, if a Tasklet exhausts its share of a container's resources before completing its computation, the Tasklet may be migrated to another container where it may resume—rather than be killed after a certain timeout, as is done in Google App Engine.

2.3 Federating Tasklets

Using Tasklets as a computational model requires applications to find appropriate containers to execute Tasklets as well as to propagate their excess capacities to other applications. The core architectural model is a trading service. In contrast to trading services introduced by Jini [2] or CORBA [3] that trade functional components, our trading service only trades one good: the execution of Tasklets.

Figure 3 illustrates the involved components and the basic messages. The Tasklet Trading Service (TTS) mediates between offers and requests for computational capacity. Tasklet containers advertise their available resources, either (1) as a Tasklet capacity, for services that are metered by resource usage or (2) as a time window with a deadline, e.g., for the termination time of an EC2 instance running a Tasklet container in the excess of its hour, as well as the cost of using the Tasklet container. Computing capacity is denoted in chunks and costs depend on the economic model that is realized by the Tasklet federation (see next section). If a Tasklet container offers execution of Tasklets on third parties, capacities for the third-parties can be propagated as well.

Figure 3 depicts the messages exchanged between the involved entities. Candidate containers are represented with enough information for applications to contact them, e.g., an IP address and port number, along with additional information like service provision contracts. An application queries the Tasklet Trading Service in order to obtain references to containers that can compute a Tasklet. Once the application chooses a Tasklet container, it interacts directly with the Tasklet container. The Tasklet container updates its capacity at the Tasklet Trading Service as it accepts Tasklets.

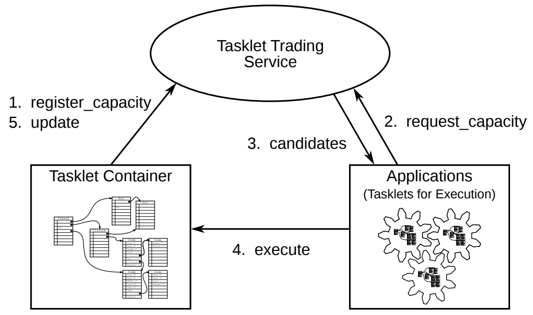

Figure 3: Tasklet Trading Service

Below are Tasklet Trading Service messages:

1. handle = register_capacity(capacity, deadline, price)
   A Tasklet container registers its Tasklet capacity and its computing deadline to the TTS. A handle is returned which is used to change the information if needed. If the container charges for execution, a price per chunk execution can be indicated.

2. candidates = request_capacity(capacity, cost_limit, info)
   An application requests computing capacity. The application approximates the necessary computing time by analyzing the Tasklet's chunks or consulting a history of prior runs. A cost-limit can be provided. Further information, e.g., for specifying a specific Tasklet container where a contract exists, can be provided in the info field.

3. result = execute(Tasklet, fee)
   After receiving a candidate list the application can choose a specific container for the execution. Based on the economic model, a fee can be provided to the Tasklet container.

4. update(handle, capacity, deadline, price)
   After capacity is consumed, in total or in part, or when the associated time window expires, the Tasklet container can update the information at the Tasklet Trading Service. If all the capacity is used the entry can be deleted (by publishing 0 chunks of computing time). Also, containers may adjust the price in order to sell excess capacity to increase revenue.

3. TASKLET ECONOMIES

In addition to the mechanics of offloading Tasklets, an economic model is needed to define the incentives and rules for offering cloud resources to others. We are in the process of implementing and analyzing different economic models for Tasklets. In the following, we present four specifically promising models and discuss them briefly.

3.1 Capacity as a Free Good

The simplest economical model for sharing capacity is to give resources away for free. Although this may not look feasible at first, there are a number of settings where the model may lead to better utilization of resources. First, in a private cloud setting one can assume that resources will not be monopolized by users. Second, even in public cloud settings the overall system can benefit from free resource sharing. In settings where users do not typically require all the local resources, e.g., due to their usage pattern or time zone differences, Tasklets can help smooth bursts of computation. As long as bursts are not clustered, users gain a better overall system experience.

Clearly, the free goods model does not provide a means to cope with free riders that extensively use other users' capacities and do not share their resources. To reduce the damage, users can secure their resources by simply running the Tasklet container with a low priority. Still, more sophisticated solutions for avoiding free riding is desirable.

3.2 Reciprocal Sharing Of Capacity

Experiences in file sharing show that free riders (e.g., in Gnutella [4]) do exist. In file sharing, upload/download ratios were introduced in order to force users to share resources. The easiest way to enforce cooperation is to introduce a simple counter that represents the number of Tasklets that were executed for another user. In order to provide an incentive, a user may offload tasks to other containers so long as she reciprocates accordingly. More precisely, Tasklets of a user u are only allowed to be executed at a remote Tasklet container if the ratio between the Tasklets that u's container has executed on behalf of other users, o, and the Tasklets that u has remotely executed, r, is greater than a threshold, t. In other words, if $o/r > t$ holds the user may execute Tasklets with other users' containers.

Clearly, local storage and manipulation of o and r can allow tampering by users. However, since the execution of Tasklets is mediated by the Tasklet Trading Service, it can take the role of a trusted third party and calculate and sign o and r.

A disadvantage of this approach is that application can only request external computing capacity if their contribution ratio is above the threshold. If an application encounters the need for additional computation it has no chance to increase the ratio and thus cannot benefit from the Tasklet federation.

3.3 Capacity Sharing Based on Virtual Currency

The previous approach can be extended by introducing a virtual currency. Rather than track the number of locally and remotely executed Tasklets for a given user, the TTS can instead manage a virtual currency. For every execution of a remote Tasklet, the Tasklet container receives a virtual currency c. c can be simply increased for each Tasklet execution, or it could reflect more finely grained models, e.g., the number of executed chunks of a Tasklet. Instead of increasing the contribution ratio, an application can "save" the received c's and use them for later remote execution.

Still this extension suffers from two problems. First, if container i executes a Tasklet for container j the currency c_j that i receives is a debt from j. Thus, without further conventions, no other container will accept c_j but container j. Second, if an application can only execute Tasklets remotely up to its savings this is a restriction. If there is need for additional computing power without enough savings, remote Tasklet computation cannot be used. An obvious extension is the use of c as currency. If a Tasklet container k accepts c_i for executing a Tasklet from an application $i \neq j$, c starts to resemble a currency. If applications can spend more c than they earned to far, credits are introduced in the system. This can be done in a simple, trust based model or real economies can be established, e.g., including interest rates and checking for creditworthiness.

3.4 Capacity Sharing Based on Real Currency

Mapping virtual currency to real currency enables new business cases. For example, one possible case is a class of generic services where providers host Tasklet containers offering Tasklet execution as a fine grained computational model. Other models are specific services, like the image processing or speech recognition services mentioned earlier.

4. RELATED WORK

Cloud computing has gained much attention over the past years, from academia as well as from service providers. Below, we discuss different cost and execution models previously proposed.

4.1 The Cost of the Cloud

One of the refreshing aspects of cloud computing, in contrast to privately-run data centers, is that operational costs are well-known, documented, and studied. For example, the CloudCmp Project [5] analyzes four different cloud providers in an attempt to guide consumers to the correct provider. Goiri et al. [6] derive profit equations for cloud providers that take into account fixed private cloud resources (that can be shut down or "insourced"/rented to others) and public cloud resources used at a specific cost rate. Given certain assumptions on power and costs, Goiri and co-authors find it better to under-provision fixed resources, use the cloud for bursts, and then insource when the fixed cluster of nodes is not busy.

Briscoe and Marinos argue for a Community Cloud [7] made out of P2P nodes to avoid "necessary evils" like vendor lock-in, cloud downtime, and privacy problems. Their work includes an idea of "community currencies" where nodes trade resources in terms of a fungible currency. A node may run a surplus, which it can then spend against nodes that run a deficit.

Durkee [8] argues that while the competition between cloud providers may reduce the cost of cloud computing, cloud computing will never be zero-cost. Cloud vendors operate in a environment of perfect competition (like cellphone vendors and airlines), so they have to make their profit by obscuring details (e.g., performance, SLAs) or costs (incoming bandwidth costs). Durkee makes the point that enterprise systems need more reliability and performance guarantees, and predicts that we will likely end up with a Cloud 2.0 that gives more value, albeit at a higher unit cost. Our work may be seen as one step towards a Cloud 2.0 charging model.

4.2 Execution Model

In contrast to recent projects like MAUI [9] and CloneCloud [10] and older work in mobile code systems [11] that concentrate on offloading code from smartphones to cloud servers for execution speed, our work concentrates on creating an accounting system to expand the available programming models in the cloud. Adaptive mobile systems, like Odyssey [12], Puppeteer [13], and Chroma [14] outline the various ways an application may be decomposed in a client/server system; such work influences how we may decompose an application into Tasklets. Flinn et al. propose [15] methods for choosing when to offload code. A node in a Tasklet system may use such information as an input to its own economic policy.

Our work may make use of cloudlets [16] to offload Tasklets to local compute resources rather than directly to the cloud. However, Tasklets are much lighter weight than the proposed cloudlet implementation of copying virtual machines between mobile nodes and cloudlet nodes [17].

Mesos [18] enables sharing of data center resources among different frameworks (e.g., MPI, Dryad, MapReduce, or Hadoop). Mesos works by offering each toolkit resources and incentivizes the framework to accept only the resources it needs. Our work may benefit from the incentive structures that Mesos introduces.

Finally, Google App Engine and Amazon EC2 are expanding offerings from the pure models we mention in Figure 1. EC2 Spot Instances [19] enable users to bid on excess capacity that would otherwise be wasted by Amazon, providing a way of lowering the cost of EC2 VMs. Similarly, App Engine offers Task Queues [20] for long running tasks that do not fit in a request/response format. Both offerings speak to the need for additional cloud computing models, of which Tasklets is one alternative.

5. CONCLUSION AND NEXT STEPS

In this paper, we proposed a lightweight computation model for using cloud resources. We developed Tasklets that form a closure of computation containing byte code as well as an interoperable representation of data required for computation and passing results. Tasklets are comprised of chunks to enable migration between containers as well as to allow for simple accounting.

We have implemented Tasklet containers in both Python and JavaScript, enabling support for a wide range of platforms. Currently, we are building a prototype Tasklet Trading Service to start integrating several of the economic models outlined in Section 3. We plan to explore how different applications, including computationally intensive interactive tasks like photo editing work in a Tasklet-based environment in contrast to a strict client/server or client-only environment. Our next steps are to explore the suitability of Tasklets in modeling and implementing applications as well as the effect of the economic models on resource utilization.

6. ACKNOWLEDGMENTS

Portions of this work is sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan.

7. REFERENCES

[1] Justin Mazzola Paluska, Hubert Pham, and Steve Ward. Structuring the unstructured middle with chunk computing. In *HotOS*, 2011.

[2] Sun Microsystems. Jini component system. `http://www.jini.org`, 2003.

[3] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.5 edition, September 2001.

[4] Daniel Hughes, Geoff Coulson, and James Walkerdine. Free riding on gnutella revisited: the bell tolls? *IEEE Distributed Systems Online*, June 2005.

[5] Ang Li et al. CloudCmp: comparing public cloud providers. In *ACM IMC*, 2010.

[6] I. Goiri, J. Guitart, and J. Torres. Characterizing cloud federation for enhancing providers' profit. In *IEEE CLOUD*, 2010.

[7] G. Briscoe and A. Marinos. Digital ecosystems in the clouds: Towards community cloud computing. In *IEEE DEST*, 2009.

[8] Dave Durkee. Why cloud computing will never be free. *Commun. ACM*, 2010.

[9] Eduardo Cuervo et al. MAUI: making smartphones last longer with code offload. In *MobiSys*, 2010.

[10] Byung-Gon Chun et al. CloneCloud: elastic execution between mobile device and cloud. In *EuroSys*, 2011.

[11] A. Fuggetta, G. P Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, May 1998.

[12] Brian D. Noble et al. Agile application-aware adaptation for mobility. In *SOSP*, 1997.

[13] Eyal De Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *USENIX Symposium on Internet Technologies and Systems*, 2001.

[14] Rajesh Krishna Balan et al. Tactics-based remote execution for mobile computing. In *MobiSys*, 2003.

[15] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *HotOS*, May 2001.

[16] M. Satyanarayanan et al. The case for VM-Based cloudlets in mobile computing. *IEEE Pervasive Computing*, October 2009.

[17] Adam Wolbach et al. Transient customization of mobile computing infrastructure. In *MobiVirt*, 2008.

[18] Benjamin Hindman et al. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[19] Amazon. Amazon EC2 spot instances. `http://aws.amazon.com/ec2/spot-instances/`, 2011.

[20] Google. The task queue python api. `https://developers.google.com/appengine/docs/python/taskqueue/`, 2012.