Design and Implementation of a P2P Cloud System

Ozalp Babaoglu
babaoglu@cs.unibo.it

Moreno Marzolla
marzolla@cs.unibo.it

Michele Tamburini
miketambu@gmail.com

Università di Bologna, Dipartimento di Scienze dell'Informazione
Mura A. Zamboni 7, I-40127 Bologna, Italy

ABSTRACT

Cloud Computing has gained popularity in both research and industrial communities. Cloud users can acquire computing resources on a need basis, achieving on demand scalability; Cloud providers can maximize resource utilizations of datacenters, increasing their return on investments. While Cloud systems are usually hosted in large datacenters and are centrally managed, other types of Cloud architectures can be imagined. In this paper we describe the design and prototype implementation of a fully decentralized, P2P Cloud. A P2P Cloud allows organizations or even individual to build a computing infrastructure out of existing resources, which can be easily allocated among different tasks. We focus on the problem of maintaining a coherent structure over a set of unreliable computing resources. We show that gossip-based protocols can be used to maintain an overlay network on top of the computing nodes, and to partition the set of resources into multiple slices in such a way that the failure of individual nodes do not compromise the overall structure. Resource partitioning is one of the most important features of a Cloud, and therefore must be supported efficiently and reliably on any Cloud architecture. We describe a prototype Java implementation that is being developed to demonstrate the effectiveness of the proposed approach.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer Communication Networks—*Distributed Systems*; D.2.0 [**Software**]: Software Engineering—*General*

General Terms

Design, Algorithms

Keywords

P2P Systems, Cloud Computing, Gossip Algorithms

1. INTRODUCTION

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

Cloud Computing has attracted enthusiastic interest from both the research community and commercial world. From the users point of view, Cloud computing provides the illusion of unlimited and on-demand scalability. In [16] the following essential characteristics of a Cloud are identified:

On-demand self service; the ability to provide computing capabilities (e.g. CPU time, storage) dynamically, as needed, without human intervention;

Network access; resources can be accessed through the network using standard mechanisms (for the most part, the HTTP protocol);

Resource pooling; virtual and physical resources can be pooled and assigned dynamically to clients according to their demand;

Elasticity; resources can be provisioned dynamically in order to enable a customer application to scale up and down quickly;

Measured service; Cloud resource and service usages are optimized through a pay-per-use business model.

Service models define the level of abstraction at which a customer interfaces a Cloud Computing environment. These are the "Software as a Service" (SaaS) model, the "Platform as a Service" (PaaS) model, and the "Infrastructure as a Service" (IaaS) model. In a SaaS Cloud, the capabilities provided to a Cloud customer are application services running on the Cloud infrastructure; the Cloud customer has no control over the infrastructure itself. Google Apps [1] is an example of a SaaS Cloud. In contrast, the capabilities provided by a PaaS Cloud consist of programming languages, tools and a hosting environment for applications developed by the Cloud customer. The PaaS Cloud user develops an application that can be executed in the Cloud and made available to service customers; development is carried out using libraries, APIs and tools possibly offered by some other company. Examples of PaaS solutions are AppEngine by Google[2], Force.com from SalesForce [3], Microsoft's Azure[4] and Amazon's Elastic Beanstalk[5]. Finally, a IaaS Cloud

[1] `http://www.google.com/a`
[2] `http://code.google.com/appengine/`
[3] `http://www.salesforce.com/platform/`
[4] `http://www.microsoft.com/windowsazure/`
[5] `http://aws.amazon.com/elasticbeanstalk/`

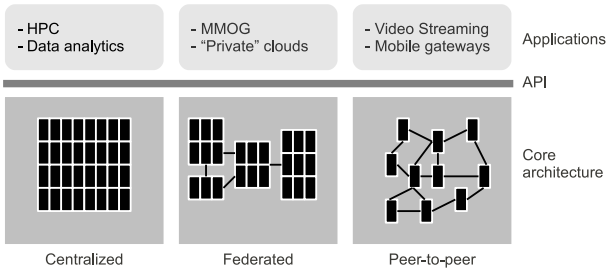

Figure 1: Cloud Computing Dimensions

provides its customers with fundamental computing capabilities such as processing, storage and networking where arbitrary software, including operating systems and applications, can be run. One of the earliest examples of IaaS Cloud is Amazon EC2[6]. In this paper we deal with the IaaS service model.

From the service providers point of view, Clouds are based on conventional computing clusters: Cloud providers invest significant resources into large datacenters, each of which is centrally managed. Building and operating a Cloud datacenter is expensive [9], so only large companies can afford such a huge investment. However, the current centralized approach to Cloud computing is not the only possibility, and in some cases might not even be the optimal choice. In [17] the authors describe a spectrum of possible Cloud architectures: centralized, federated and Peer-to-Peer (P2P) (see Figure 1).

Centralized Clouds constitute the current commercial offerings. The centralized model is appropriate for applications such as scientific computations, data mining, Internet-scale Web Services and delay-sensitive applications that cannot tolerate high communication delays. Federated Clouds are a logical evolution of the centralized approach: they involve multiple Clouds that are tied together to build a larger one. Federation can be used to enhance reliability through physical partitioning of the resource pool, and also to address communication latency issues by binding clients to the "nearest" datacenter. Furthermore, federated Clouds are an interesting alternative for those companies who are reluctant to move their data out of house to a service provider due to security and confidentiality concerns. By operating on geographically distributed datacenters, companies could still benefit from the advantages of Cloud computing by running small Clouds in-house, and federating them into a larger Cloud. Finally, by stretching the idea of federated Clouds to the extreme, we can build a Cloud out of independent resources that are opportunistically assembled. Such P2P Clouds could be built by assembling individual peers without any central monitoring or coordination component. P2P Clouds can enable provisioning of resources at low or no cost; loosely-coupled distributed applications where the physical location of nodes is important to keep data/computation near the end user, can benefit from the P2P model. The Cloud API provides an interface for resource negotiation, allocation and monitoring, regardless of the specific Cloud architecture.

[6] `http://aws.amazon.com/ec2`

The case for a P2P Cloud. While P2P Clouds are unlikely to provide the features and Quality of Service (QoS) guarantees of a centralized or federated Cloud, there are nevertheless some usage scenarios for which a fully distributed Cloud architecture can be useful. A P2P Cloud can be assembled at virtually no cost using existing resources; therefore, many small or medium-sized organizations could turn idle resources into a computing infrastructure which can be partitioned among a number of internal "customers". For example, an engineering company could partition its spare resources (desktop PCs) among internal groups, e.g., the project team to perform structural simulations, the IT group to analyze network access logs for intrusion detection, and the accounting group to compute cash flow and other financial indicators for evaluation purposes. According to the needs of the various groups, it could become necessary to shift more resources towards a specific team (e.g., when approaching a project deadline, the engineering team would get more computing power to finish the calculations). A P2P Cloud would provide on-demand scalability, access to computing and storage space with no single point of failure nor central management. New resources can be added to the pool by simply installing a software daemon on them. Some applications which can be executed on a P2P Cloud include [1] embarrassingly parallel computations, multimedia streaming, online gaming requiring low latency and a high level of interactivity, collaboration tools with shared data.

P2P Clouds vs Volunteer Computing. Volunteer Computing (VC) is a well known computing paradigm, where users execute third-party applications. VC systems usually require users to install a specific application on their PC; the applications fetches and processes input data from a central location, and uploads the results; VC systems are mainly targeted at embarrassingly parallel scientific applications. The widely used BOINC system [4] separates the client program from the application-specific part: users install the BOINC client and select the project(s) they support. Examples of projects running on the BOINC platform are SETI@home[7] (analysis of radio signals from space to detect potential extra-terrestrial emissions), Folding@home[8] (protein folding), Einstein@home[9] (gravitational wave detection), and many others. After a project has been selected, the BOINC client fetches and executes the specific application (task).

Clouds and VC systems have some important differences since they serve different purposes; the differences are summarized in Table 1. The resources of a Cloud are generally owned by a single entity (the Cloud provider), while VC relies on resources provided by third parties. A IaaS Cloud provides a virtualized environment where arbitrary guest OSes and applications can be executed; on the other hand, systems like BOINC are only capable of executing specific applications running inside the client. Clouds ensure a high level of QoS in order to remain competitive; VC systems, on the other hand, cannot provide any guarantees since all computing nodes are managed by individual users, and can be shut down at any time. Clouds are hosted on large datacenters, which can be federated to improve reliability, while VC

[7] `http://setiathome.berkeley.edu/`
[8] `http://folding.stanford.edu/`
[9] `http://www.einstein-online.info/`

IaaS Cloud	Volunteer Computing	P2P Cloud
Single resource provider	Multiple resource providers	Multiple resource providers
Virtualized environment	Runs specific applications	Virtualized environment
High reliability	Unpredictable reliability	Unpredictable reliability
Local or Geographic scale	Geographic scale	Local or Geographic scale
Public, private or hybrid	Public	Public, private or hybrid

Table 1: Comparison of IaaS Clouds, Volunteer Computing and P2P Clouds

systems are geographically distributed with some centralized control (the task and data repositories). Clouds can be public, private or hybrid: public Clouds provision resources to the general public, private Clouds operate only for a single organization (which is typically the Cloud owner), and hybrid Clouds combine a public part with a private part. VC infrastructures are generally public only, in the sense that the computing resources can in principle be used by any project. P2P Clouds borrow features both from IaaS Clouds and from VC systems (see Table 1). A P2P Cloud differs from a VC system because there is no central coordination nor central repository of tasks.

Our Contribution. In this paper we describe the architecture and prototype implementation of Peer-to-Peer Cloud System (P2PCS), a fully distributed IaaS Cloud infrastructure. In particular, we focus on algorithms and protocols for (i) maintaining cohesion over a set of unreliable peers, and (ii) partitioning (slicing) the resources into multiple subclouds that can be assigned to individual users. P2PCS builds on top of several gossip-based algorithms (Peer Sampling Service [15], Slicing Service [11], T-Man [14] and others) which together implement robust and scalable high level Cloud operations.

This paper is organized as follows: in Section 2 we briefly review the state of the art with respect to P2P Clouds; in Section 3 we describe the system model; in Section 4 we describe the architecture of P2PCS and describe its main components; in Section 5 we describe a Java prototype implementation of P2PCS; finally, conclusions and future research directions will be discussed in Section 6.

2. RELATED WORK

In recent years, several authors have recognized the potential benefits of P2P Cloud architectures. In [6] the authors sketched a general-purpose framework to support fully distributed applications running over a very large-scale and dynamic pool of resources. The authors list several gossip-based protocols that can be applied to form the subclouds and to implement bootstrapping, monitoring and control services. Building on the main idea of [6], in this paper we present a practical architecture, with a prototype implementation, of a P2P Cloud.

A different proposal for a distributed Cloud architecture is given in [8, 7]. The authors present Cloud@Home, a hybrid system which combines features from the VC model and Cloud computing paradigm. It should be observed that the Cloud@Home architecture relies on centralized components, while allowing end users to contribute additional resources. On the other hand, our proposal is fully decentralized, and does not require any central bookkeeping service. At the time of writing we are not aware of any implementation of a Cloud@Home prototype

In [18] a different direction is taken: the authors propose a distributed computing platform called Nano Data Centers (NaDa). NaDa uses home gateways, controlled by ISPs, to provide computing and storage services. Using a managed peer-to-peer model, NaDa form a distributed data center infrastructure.

We finally mention Wuala[10] as an example of Cloud based storage service. Wuala allows users to trade space on their hard disks to receive encrypted chunks of files uploaded by other users. Another distributed Cloud storage system is described in [19]. These are purely storage services, therefore they offer no support for executing computational tasks. Our architecture, on the other hand, aims at providing both computation and storage services.

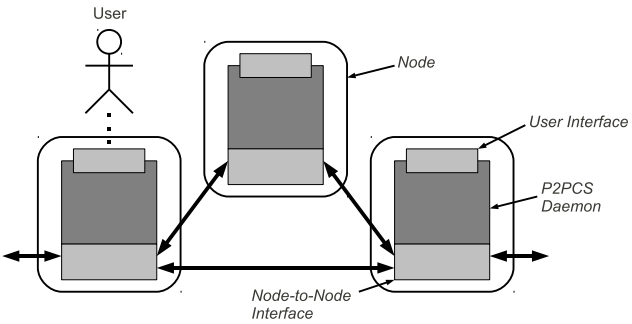

Figure 2: System Model

3. SYSTEM MODEL

We consider a large set of networked nodes which can be owned by different individuals or organizations. Each node includes a processor, RAM, storage space and network connectivity; we do not require that all nodes are homogeneous.

Users of this system (which in general are the owners of the nodes) share the resources (CPUs, memory, disks) cooperatively. To do so, they install a software daemon on each node (Figure 2) which takes care of maintaining cohesion and gracefully handle churn; in fact nodes are not required to be reliable, so they can join or leave the system at any time. The software daemon has two separate interfaces: a *user interface*, through which users can inject requests into the system, and a *node-to-node interface* which is used to communicate with other peers.

The most important operation provided by the P2P Cloud system is the management of *slices* (partitions). A user can request a fraction of the available resources matching a given query (e.g., the 200 nodes with fastest processor). The system checks whether the query can be satisfied, and if so allocates the node to the requester. Therefore, at any time the global Cloud may contain multiple disjoint sub-Clouds

[10] `http://www.wuala.com/`

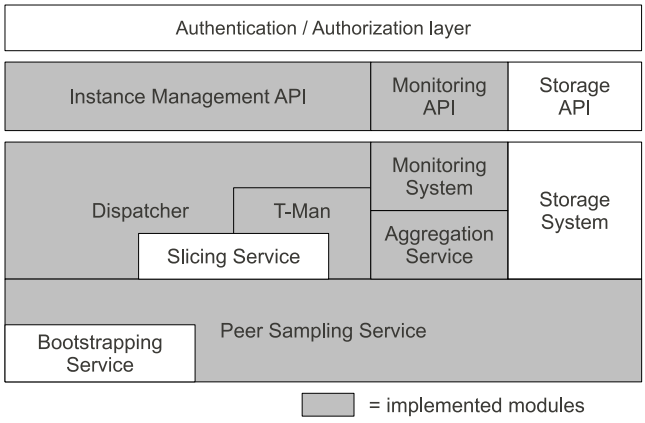

Figure 3: Layered architecture of the P2PCS; the shaded modules have been (at least partially) implemented in the prototype.

assigned to users. Slices are dynamic, since users may request their partitions to grow or shrink.

Once a slice has been setup, the owner can upload and execute applications or Virtual Machine images on the nodes. The API exposed by the user interface is similar to a conventional IaaS Cloud API, such as Amazon EC2 [3] or Amazon S3 [2]. Some practical examples of API functions implemented in the P2PCS prototype will be illustrated in Section 5.

We remark that the nodes are managed by their respective owners, hence no QoS guarantee can be provided. Application failures resulting from node crashed must be handled by the user running the application (this is what happens with conventional IaaS Clouds as well). However, P2PCS ensures cohesion of both the global Cloud and all slices: this means that even in case of multiple failures, the surviving nodes are still part of their slice and can interact with other peers in the slice and in the global Cloud. Borrowing the analogy used in [6], we consider a P2P Cloud as a real cloud made of many water droplets: while the cloud is constantly changing since as individual drops join or leave, it always has an well defined shape. In the same way, a P2P Cloud is made of a mutable set of resources which are kept together using gossip-based, epidemic protocols.

4. P2PCS ARCHITECTURE

In this section we give a high-level description of the P2PCS architecture. We focus on algorithmic and protocol issues; additional details will be given in Section 5.

As already discussed, P2PCS is implemented as a collection of identical interacting processes, each one running on a separate host. Each process is made of several software modules that are roughly organized according to the layered structure shown in Figure 3.

The **Peer Sampling Service (PSS)** [15] aims at providing each node with a list of peers to exchange messages with; this is achieved by maintaining an unstructured overlay over the set of peers. The PSS is implemented as a simple gossip protocol, as follows. Each node maintains a list of $k > 0$ neighbors, called the *local view*; each element in the local view contains the ID (e.g., IP address) of a neighbor and a timestamp indicating when that neighbor was first added into the local view. Periodically, neighbors exchange and merge their local views, removing the oldest entries so that the number of neighbors remains equal to k. The local views are dynamic, since at each message exchange a new view is constructed. Therefore, the set of neighbors of each node is constantly changing, resulting in a dynamic random graph overlay. Using the simple gossip protocol just described, the PSS can keep the overlay connected also in presence of churn, i.e., nodes joining and leaving the system [15]. This feature is fundamental in a dynamic environment where resources are managed by individual users. The PSS uses the **Bootstrapping Service** [13] to gather an initial set of nodes to start the message exchange. The Boostrapping Service is used, as the name suggests, to "cold boot" the system, since at the beginning each peer does not know the identity of other nodes in the Cloud; in the current prototype the bootstrap is performed from a file containing the IP addresses of all nodes.

The **Slicing Service (SS)** [11] is used to rank the nodes according to one or more attributes. This service is used to request slices of the whole Cloud according to some user-defined criteria, e.g., a fraction of 5% of the total number of nodes, the top 1% fastest nodes, and so on. When a user requests the allocation of some nodes according to a specific metric (multi-attribute metrics can be supported as well), the SS ranks the nodes according to that metric, and returns the set of resources matching the query. In the current prototype, the SS only supports the creation of slices with a user-defined number N of randomly chosen nodes which are not already part of some other slice.

The **Aggregation Service (AS)** [12] is used to compute global measures using local message exchanges. The AS allows each peer to know system-wide parameters without the need to access a global registry. Examples include the network size (number of nodes in the Cloud), average load, number of slices (subclouds) and so on. The AS works as follows: each node p keeps a value s_p; periodically, s_p is sent to all neighbors of p (the list of neighbors is maintained using the PSS). When a neighbor q receives the value s_p, it executes the instruction $s_q \leftarrow \text{UPDATE}(s_q, s_p)$ to compute a new local value s_q from the old value and s_p. The function UPDATE() depends on the global value which must be computed. For example if $\text{UPDATE}(x, y) := (x + y)/2$, then the protocol computes the global average of all local values; if all values are initially zero, except for one node which is assigned local value 1, then the protocol converges to $1/N$, where N is the number of peers, from which the network size N can be estimated. The **Monitoring System** is implemented on top of the AS, and collects global system parameters as illustrated above; the values computed by the Monitoring System are available to users through an API.

The **Monitoring System API** provides two operations for starting and stopping the display of run-time instance information; these operations are roughly equivalent of Amazon EC2 `ec2-monitor-instances` and `ec2-unmonitor-instances`. In the current P2PCS prototype, the monitoring interface allows a user to display the topology of the network, and the set of nodes of the slice a node belongs to. At the moment, the main use of the monitoring API is for debugging purposes.

T-Man [14] is a gossip-based protocol for building an overlay network with a given topology (tree, ring, mesh or other structures). P2PCS uses T-Man to bind together the

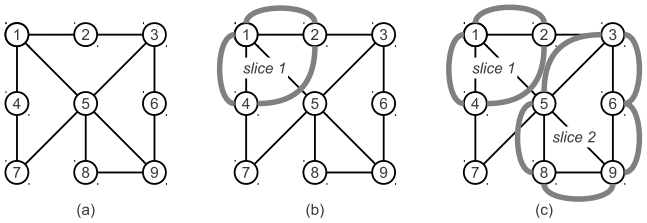

Figure 4: Crating subclouds in P2PCS

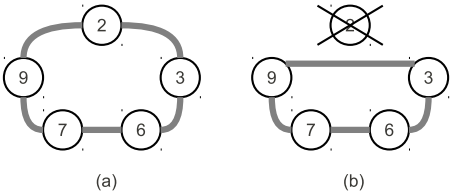

Figure 5: Slice maintenance with node failures

nodes belonging to the same slice, by linking all peers of the same slice with a separate ring overlay (separate from the random overlay maintained by the PSS). As a practical example, let us consider the situation shown in Figure 4(a); there are nine nodes, labeled 1–9, connected through a random graph overlay maintained by the PSS. Suppose that a user requests the creation of a subcloud (slice) with 3 nodes: the system selects the requested number of nodes (for example, nodes 1, 2 and 4) and creates a ring overlay such as the one shown in Figure 4(b). Each node of the slice has a direct link to its predecessor and successor. Thanks to the T-Man protocol, the ring overlay is maintained even if nodes in the slice fail: the failed nodes are removed from the ring, and links are rearranged to connect the surviving peers. Multiple slices can be active at the same time; for example, if another user requests a slice with four nodes, the system may select $\{3, 5, 6, 8, 9\}$, resulting in the situation shown in Figure 4(c).

The **Dispatcher** is responsible for handling the requests submitted by the user through the high level user interface, and translate them into the appropriate low level gossip protocol commands which are sent to the other nodes.

The **Instance Management API** contains the interface which allows a user to manage the resources instances: creation of a new instance, termination of an active instance, enumeration of currently owned resources and so on. These are similar to the operations `ec2-run-instances`, `ec2-start-instances`, `ec2-stop-instances`, `ec2-terminate-instances` and so on, provided by the Amazon EC2 service [3].

A IaaS Cloud also provides operations to deal with storage space allocation: these operations allow users to request, grow or shrink storage space as needed. In a P2P Cloud the storage service must be implemented as a fully distributed service. Several systems have been proposed in the literature (see [10] and references therein). Finally, the authentication/authorization layer is responsible for ensuring that the local node can be made available to trusted users only, should the owner decide so.

5. PROTOTYPE IMPLEMENTATION

We implemented a Java prototype of the P2P Cloud system described in the previous sections, using JRMI for remote communication management. The prototype currently implements the main features of the shaded modules of Figure 3; at this time, authentication/authorization and storage space management are not implemented. The prototype aims at demonstrating the feasibility of the idea of a fully decentralized Cloud system. While the core algorithms used in P2PCS (Peer Sampling Service, Slicing Service, T-Man) have been thoroughly analyzed in the literature, to the best of our knowledge this is one of the first attempts to create a non-trivial application using them as building blocks. This is an important result, since our experience validates the claim that complex behaviors can be engineered from the simple interactions of well understood protocols [5].

The P2PCS prototype consists of a Java servent (server-client) which runs on all hosts that are part of the Cloud. The prototype includes a set of Bash scripts that wrap the client-side Java programs which invoke various operations from the user interface API. Specifically, the following scripts are available:

- `run-nodes` *`slice_id number`* creates a slice with *number* nodes; *slice_id* is set as the name of the newly created slice. The nodes are chosen without any particular criteria;
- `terminate-nodes` *`slice_id nodeName1 ... nodeNameN`* removes the named nodes from the slice *subcloud_id*.
- `add-new_nodes` *`slice_id nofnodes`* adds *nofnodes* nodes to the slice identified by *slice_id*. The new nodes are chosen without any particular criteria among those which do not belong to any slice.
- `describe-instaces` *`nodename`* outputs a human-readable description of the given node, including the name of the slice it belongs to (if any), including the name of the neighbors according to the ring overlay defined by T-Man.
- `montor-instaces` returns the global size of the network using the Aggregation Service; the size is dynamically updated, until the `unmonitor-instances` command is invoked.
- `unmonitor-instaces` interrupts the display of the network size.

We now give some practical example. The command

```
$./startNode.sh -n node2
```

starts a servent on the local host, assigning to it the human-readable identifier "node2" (this identifier is used for debugging purposes only). Assuming we have started the P2PCS servent on 10 nodes, labeled as "node1", "node2", ... "node10", we can create a slice by requiring 5 nodes with the following command:

```
$./run-nodes.sh -n node3 mySubCloud 5
```

The command above injects on node "node3" the request to create a slice called "mySubCloud" with 5 elements. The resulting slice might contains the nodes $\{2, 3, 6, 7, 9\}$ organized as the ring shown in in Figure 5(a). (recall that the

nodes of a slice are chosen without a particular criteria, therefore the command above might select any subset of 5 nodes and arrange them in any order along the ring). If we remove node 2, either by killing the P2PCS servent or using the command:

```
$./terminate-nodes.sh mySubCloud node2
```

we obtain the situation shown in Figure 5(b). The T-Man protocol automatically detects that a node failed, and reroutes the connections of the ring overlay around node 2.

The system proved to be quite robust to failures: we allocated about 40 nodes on our computer lab and killed about half the nodes without disrupting the service; all remaining nodes were able to quickly reconfigure themselves by excluding the failed peers.

6. CONCLUSIONS AND FUTURE WORK

In this paper we described the architecture and prototype implementation of P2PCS, a fully distributed IaaS Cloud system. P2PCS uses gossip-based protocols to manage a large, unreliable resource pool without any central coordinator. We developed a Java prototype to demonstrate the main features of P2PCS: self-organization and robustness to failures. Initial results are encouraging and suggest that a decentralized Cloud infrastructure can indeed be realized using well understood techniques and protocols.

We are currently working on the remaining components of the architecture, which have been left out of the prototype: the authentication/authorization layer, the storage management service, the bootstrap service and the query-based resource selection algorithms. Finally, we plan to perform a comprehensive performance and reliability assessment of P2PCS through live experiments of the prototype on a suitably sized testbed.

The source code of the P2P Cloud System can be downloaded from `http://cloudsystem.googlecode.com/svn/trunk/source/`, and is distributed under the terms of the GNU General Public License (GPL), version 3.

7. REFERENCES

[1] Clouds and peer-to-peer. URL, June 11 2009. `http://berkeleyclouds.blogspot.com/2009/06/clouds-and-peer-to-peer.html`.

[2] Amazon. *Amazon Simple Storage Service API Reference (API Version 2006-03-01)*, Mar. 2006. Available at `http://docs.amazonwebservices.com/AmazonS3/latest/API/`.

[3] Amazon. *Amazon Elastic Compute Cloud API Reference (API Version 2011-07-15)*, July 2011. Available at `http://docs.amazonwebservices.com/AWSEC2/latest/APIReference/`.

[4] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *Proc. 5th IEEE/ACM Int. Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[5] O. Babaoglu and M. Jelasity. Self-$*$ properties through gossiping. In *Philosophical Transactions A of the Royal Society*, volume 366, pages 3747–3757. October 2008.

[6] O. Babaoglu, M. Jelasity, A.-M. Kermarrec, A. Montresor, and M. van Steen. Managing clouds: a case for a fresh look at large unreliable dynamic networks. *SIGOPS Oper. Syst. Rev.*, 40:9–13, July 2006.

[7] V. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa. Cloud@home: Bridging the gap between volunteer and cloud computing. In D.-S. Huang, K.-H. Jo, H.-H. Lee, H.-J. Kang, and V. Bevilacqua, editors, *Emerging Intelligent Computing Technology and Applications*, volume 5754 of *LNCS*, pages 423–432. 2009.

[8] V. D. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa. Volunteer computing and desktop cloud: The cloud@home paradigm. In *Network Computing and Applications, IEEE International Symposium on*, pages 134–139, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[9] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39:68–73, December 2008.

[10] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell. A survey of peer-to-peer storage techniques for distributed file systems. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, pages 205–213, Washington, DC, USA, 2005. IEEE Computer Society.

[11] M. Jelasity and A.-M. Kermarrec. Ordered slicing of very large-scale overlay networks. In *Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 117–124, Washington, DC, USA, 2006. IEEE Computer Society.

[12] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

[13] M. Jelasity, A. Montresor, and O. Babaoglu. The bootstrapping service. In *Proceedings of the 26th IEEE International Conference/Workshops on Distributed Computing Systems*, ICDCSW'06, pages 11–, Washington, DC, USA, 2006. IEEE Computer Society.

[14] M. Jelasity, A. Montresor, and O. Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009.

[15] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[16] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Special publication 800-145 (draft), Gaithersburg (MD), Jan. 2011.

[17] F. Panzieri, O. Babaoglu, V. Ghini, S. Ferretti, and M. Marzolla. Distributed computing in the 21st century: Some aspects of cloud computing. In *Dependable and Historic Computing*, volume 6875 of *LNCS*. 2011.

[18] V. Valancius, N. Laoutaris, L. Massoulié, C. Diot, and P. Rodriguez. Greening the internet with nano data centers. In *Proc. 5th Int. Conf. on Emerging networking experiments and technologies*, CoNEXT '09, pages 37–48, New York, NY, USA, 2009. ACM.

[19] K. Xu, M. Song, X. Zhang, and J. Song. A cloud computing platform based on p2p. In *IT in Medicine Education, 2009. ITIME '09. IEEE International Symposium on*, pages 427–432, Aug. 2009.