Managing Clouds: A Case for a Fresh Look at Large Unreliable Dynamic Networks*

Ozalp Babaoglu
University of Bologna
babaoglu@cs.unibo.it

Márk Jelasity
University of Bologna
jelasity@cs.unibo.it

Anne-Marie Kermarrec
INRIA/IRISA
akermarr@irisa.fr

Alberto Montresor
University of Trento
montreso@dit.unitn.it

Maarten van Steen
Vrije Universiteit Amsterdam
steen@few.vu.nl

ABSTRACT

Peer-to-peer (P2P) protocols have proven efficient to provide scalable support to many large-scale distributed applications, successfully coping with unreliability and dynamics. However, to exploit them in a wider range of environments, such as very large-scale networks of smartphones or set-top boxes, it is imperative to make P2P protocols manageable: we need to be able to start, bootstrap and stop protocols, and assign resources dynamically. In this paper we present a general-purpose framework aimed to support several fully distributed applications running independently over a very large scale and dynamic pool of resources. We call this resource pool a cloud. The basic idea of the framework is a declarative application suit description, that describes what applications should be running on what resources, and a middleware that makes sure the currently available and dynamic cloud *self-organizes* into the configuration represented by the description, creating the subclouds that are assigned to applications. The middleware also provides additional functionality, such as bootstrapping overlay networks, to support the applications. Our preliminary ideas on the implementation rely on various gossip-based protocols, that are applied to form the subclouds and to implement bootstrapping, monitoring and control services. Most of all, this position paper sets an exciting research agenda to fully exploit the possibilities offered by very large scale dynamic networks.

1. INTRODUCTION

Distributed applications running over very large scale, dynamic, heterogeneous and unreliable distributed systems are now commonplace. Examples include desktop grid environments, file sharing networks, content distribution networks, voice over IP, and various distributed gaming applications. For the most part, these applications are made possible through peer-to-peer (P2P) technologies.

Despite intense activity over the last several years, the arena of P2P algorithm design continues to be in a growing phase, with a plethora of protocols available for a wide range of functions. Yet, the current state of affairs has failed to relax one important restriction: peer-to-peer protocols and the systems that exploit them are often very specific to a given application. As a result, a file-sharing system is only adapted for file sharing, a desktop grid is able to execute only specially tailored and centrally managed tasks, and so on. This situation is somewhat comparable to having a powerful computer that can run only one application, without the possibility of reprogramming to exploit all its potentials.

Clearly, this tight coupling between systems, protocols, and applications is unnecessarily restrictive. For a given (distributed or networked) system, we should be able to freely change the P2P protocols at runtime, add new ones and stop existing ones. Furthermore, we should be able to merge systems on which P2P applications are running, or split existing ones. Such flexibility would allow us to deploy P2P applications much easier and let several such applications to co-exist independently while making use efficiently of shared underlying resources and protocols.

The major impediment to this form of deployment is the extreme scale of the underlying system. Most P2P applications have been designed with thousands, if not millions of nodes in mind. As a consequence, managing the suite of protocols and applications dispersed across a huge collection of nodes is far from being a trivial task. We take the position that such management can be achieved only if the participating nodes have the capability to automatically organize and manage themselves rather than being managed by some external entity.

To this end, we propose to make use of a declarative description of a desired application that specifies what services, components, etc., should be running on what resources. In our proposal, the underlying system of nodes and the application descriptions are completely separated: the descriptions are disseminated throughout the nodes who then *self-organize* to ensure that the desired applications are simultaneously supported. The latter requires the implementation of a lightweight and transparent distributed middleware layer. In this paper, we sketch a possible implementation based on *gossip-style* protocols. Instead of describing an existing system, we propose a rough outline and some initial results, however, most of the issues we raise represent open research issues. In the following we elaborate on the aspects mentioned above.

2. SYSTEM MODEL

*In *ACM SIGOPS Operating Systems Review*, 40(3):9–13, 2006. doi:10.1145/1151374.1151379

We consider a huge collection of networked nodes, typically owned by a single organization. We already see such collections, for example, with ISPs who place modems and set-top boxes at their customers' homes. The benefit of retaining ownership is that the ISP stays in control of the quality of hardware that provides access to their network. In the long run, the cost of retention is often lower than having to provide services to support customers with their own hardware. In a similar way, we may eventually see more telcos handing out smartphones at low costs to their subscribers, not only to bind users to their services, but also to lower management costs.

Given such a situation, we expect that these organizations would want to deploy massive services across their networks. In practice, we expect that the number of services is relatively low, say a few dozen. An important issue is that many of these services will have a strong peer-to-peer flavor. For example, one can think of collaborative streaming, gaming, and messaging, just to name a few. Likewise, considering the enormous popularity of systems such as Wikipedia and their current scalability problems, it is not difficult to envisage peer-to-peer solutions for them offered as services to customers. Furthermore, we see efforts toward building fully-decentralized collaborative personal video recorders that can be efficiently realized through set-top boxes with sufficient storage capacity [7].

The computing medium we are thus interested in consists of a very large number of computing devices that are connected through a routed network. The devices, or nodes, are unreliable. Nodes may leave or join at any time. We find it intuitive to call this environment a **cloud**: a cloud contains a huge number of water droplets or ice particles that are constantly leaving and joining, making the boundary of the cloud fuzzy, but nevertheless forming an identifiable shape. This metaphor also suggests that an individual node is insignificant but that a collection of nodes forming a cloud forms a consistent structure. Indeed, the abstraction of a cloud of nodes will be considered as a *practically infinite, continuous medium*. Note that this environment may also be composed of a collection of clouds.

One of the key feature of peer to peer systems on which scalability relies is resource aggregation. The resources in the system (such as storage space, memory or CPU cycles) are the sum of the resources available at the individual nodes. The only type of resource that is assigned to applications is a cloud of nodes. The assigned cloud is selected based on certain attributes relevant for an application using it; for example, we can select x% of nodes or N nodes based on storage capacity, availability, bandwidth, etc. The unit of allocation in our scheme is an entire node in the sense that it is not possible to allocate portions of it to applications — a node either *participates* in facilitating a given application or it does not.

The usage of the word cloud in the context of resource assignment is, again, intentional. The collection of nodes that we assign to an application is generally very large and highly dynamic. Obviously, the system dynamism is reflected in the cloud composition. This is possible since the (peer-to-peer) applications we intend to support are designed with such environments in mind.

3. APPLICATION SUITE DESCRIPTION

The purpose of an application suite description is to define what applications are running on the network, and what resources should be assigned to these applications. A description is independent of the actual available set of nodes, in that it does not refer to any individual nodes. Instead, it identifies resources as subclouds that span a certain proportion of the entire cloud and that possibly have some desirable properties such a given reliability or storage availability.

To give a simple example, we can have a description where we assign the most reliable 10% of the network to a monitoring/control service, and divide the remaining 90% randomly and equally among two user applications. Since the set of nodes keeps changing, as well as their characteristics, the top 10% most reliable nodes will keep changing as well, although probably more slowly (after all, they are the most reliable ones). The monitoring service and the two applications in our example are responsible for dealing with any churn or unreliability of their respective subcloud. The only thing that we require from the middleware layer that lets the system self-organize to meet the actual description is that it ensures at any time that the 10% most reliable nodes are always assigned to the monitoring service (perhaps invoking any join or leave protocols, etc) and the rest of the nodes are divided equally among the applications.

The description itself is allowed to change abruptly as well. In fact, this is exactly the mechanism to stop and launch services and applications: when a new application needs to be launched, it is simply added to the description which triggers the self-organizing middleware to assign resources and launch the application. Similarly, stopping a service and recycling its resources, or simply shifting priorities (amount of resources assigned to applications) involves updating its description. Individual applications will experience only some dynamism of the subcloud they are running on as a function of these global changes, but otherwise, they are not concerned or know about global resource assignment issues.

4. IMPLEMENTATION

Evidently, the abstractions and the usage scenarios sketched above raise many challenging implementation issues. For example, what does "changing the suite description" mean? Where this description is stored? How do applications get launched and how are the prescribed resource assignments maintained?

In short, the task is to implement the middleware that connects two entities: the cloud on which the system as a whole is running and the possibly-changing system descriptions. This means that all participating nodes have to know what application(s) to run at any time, and proper startup, node joining and node leaving have to be managed for all applications using the procedures the given application prescribes.

4.1 The Challenge

This task is very challenging because we do not assume the existence of a specific infrastructure: the system has to work completely automatically, in a self-organizing way due to its large scale and dynamic behavior. The only possibil-

ity for human intervention is by changing application suite descriptions; manually mapping descriptions to the current network is simply not feasible.

In principle, by investing in infrastructure (such as powerful servers), it would be possible to control the system from a central location. This, however, is an expensive and vulnerable solution that we wish to avoid. Yet without central control, things like assigning nodes to applications based on descriptions is highly non-trivial since it requires global agreement among the nodes. Effectively, a relevant mapping between clouds and suite description relies on some form of global assessment of the system whereas nodes have only a very limited knowledge of the network. The fact that nodes continuously join and leave only further complicates the situation.

4.2 The Cloud

The cloud has been used as a metaphor for a very large scale and dynamic group. In practice, we expect clouds to be dynamically and automatically formed as nodes belonging together get to know each other. To actually implement such a cloud, we propose to rely on a *peer sampling service* [2]. This service provides random samples from the cloud to each participating node. The implementation of this service is a gossip-based protocol that periodically exchanges and updates a list of random members at each node. This protocol provides high tolerance to rapid changes in the system, including failures, which makes it attractive as a minimal but very robust bottom layer for our architecture.

More generally, similar gossip-based protocols can be used in a subcloud as well, so that related nodes (according to the application suite description) get to know each other.

4.3 Application Suite Description: Implementation

We do not address language issues that relate to the description of an application suite. We intentionally work with an informal concept that nevertheless allows for a list of applications, and the description of the required allocation of resources to the respective applications. This resource-allocation description expresses what proportion of the nodes should be assigned to a given application, and optionally a description of desirable properties of these nodes. Constraints on the allocation could take the form of describing a minimal number of nodes, or a minimal number satisfying a certain property such as a node's average availability.

We assume that the application suite description is known to all participating nodes at all times. This can be achieved by piggybacking an anti-entropy gossip protocol on the peer sampling service implementation, which maintains the description at all nodes. This can be very cheap since we expect the description to be very compact, and in most of the cases, the piggybacking does not result in observable increase of traffic due to the small size of packets that are being exchanged.

Access rights to update the description have to be implemented as well, which can be done using common techniques such as public key cryptography and signed certificates.

4.4 The Middleware Service

Having addressed how the cloud and the application suit description can be implemented, we move on to the middleware layer that is responsible for keeping the system in a state that corresponds to the description.

The services that this middleware has to perform are the following:

Slicing: Responsible for assigning the right proportion of nodes (subclouds) to applications, taking care of other possible requirements about these nodes,

Bootstrapping: Responsible for starting up an application from scratch,

Churn handling: Responsible for assisting the application in handling churn and failures.

We describe these middleware services one-by-one below.

4.4.1 Slicing

As mentioned above, the application suit description assigns resources to applications, and "resource" always means a percentage of the total nodes, that may or may not have special characteristics. Since all nodes are assumed to have a copy of the description, the problem boils down to letting the nodes select which subcloud they want to belong to. The simplest case is when the network is partitioned at random, and no special requirements are placed over the nodes in any partition. In this case, each node can pick a subcloud at random, proportional to the required size of the subcloud.

If the partitions have other constraints, for example, we need to create a partition from the most reliable nodes, then the problem becomes much more challenging. We have preliminary results that point towards this direction [3]. Those results show that in a few cycles, the network can get partitioned according to the description.

The approach that we have taken is that each node gets an associated random ID from $[0, 1]$. Using a gossip-based protocol, nodes keep swapping their random ID until the order of IDs reflects the order of a given gradient (bandwidth, storage capacity, etc.). This way, if a node finally gets an ID which is, for example, less than 0.5, then it knows it is in the top half of the network according to the gradient. In this manner partitions can be selected locally.

Partitions created this way need to form a cloud themselves. This can be achieved if the contact nodes are also disseminated along with the application suite description: each node in each application keeps publishing itself as a contact node for its own application, using the anti-entropy epidemic database update propagation protocol. This way each node has constantly updated information about possible contacts to all applications. New nodes use these contact nodes to join the peer sampling overlay we described above, hence connecting to their subcloud.

4.4.2 Bootstrapping

Bootstrapping is a service that can be issued for a specific subcloud to build an overlay topology, which can in turn be used by the application or service that occupies that subcloud. It will be issued when the application that needs a specific overlay topology (such as a specific distributed hash

table) is being started for the first time, or when dramatic events happen such as splitting or joining two systems. We assume that topology is maintained through an appropriate P2P protocol; the bootstrapping service only provides quick and cheap startup. This is crucial if we want to allow the system to be truly dynamic and support the launching of possibly complex applications quickly, to be run perhaps for a short time.

We have developed a number of protocols for bootstrapping different topologies [1, 6, 5]. These implementations are also gossip-based, and therefore are simple and robust.

4.4.3 Churn

The churn-handling service is rather simple: it is responsible for adding nodes to and removing nodes from an application. The service utilizes join and leave methods implemented by the application, by calling them at the appropriate times. For example, if a new node joins the system, and decides to be part of a particular subcloud (via the slicing service), then it needs to first get the application code from one or more contacts from the subcloud, and subsequently issue a join method that is available in the obtained code. When leaving, a node notifies the churn-handling service about this fact, which then invokes the leave method registered by the application.

Dealing with crashes is the responsibility of the application itself. The middleware layer itself will be extremely robust to failures, but applications also have to be robust themselves, because their subcloud inherits any unreliability in the available set of nodes.

4.5 Usage Scenario

Using these middleware services, the scenario for the lifecycle of a user application would be the following:

1. The user writes the application against an API that allows it to read the application suit description and gives access points to the middleware such as leave and join methods, etc.
2. The user is assigned a subcloud via an update to the application suit description, through a procedure we do not detail here. It can be automatic, based on access rights, or done through a central administration point.
3. The system will then automatically self-organize into the new configuration that contains the new subcloud, and at least one node, that of the user. This node will seed the spreading of the application code over the subcloud in an epidemic fashion, or through more sophisticated content distribution algorithms, or a combination of the two techniques. Those nodes that are assigned to the application but have not received the application code will actively try to get it from their contacts in the cloud.
4. In parallel with the startup, the bootstrapping service will build any necessary overlay networks and present them to the application through an API.
5. Removing the application simply means removing it from the application suit description. As a result, all nodes will get automatically re-assigned through the slicing service and any old application state simply gets discarded.

Executing long-running applications as generic services is also possible in an identical fashion.

4.6 Federation and Splitting

It might become important or useful to split or merge complete systems that follow the present framework. This is rather straightforward. Recall that the system is composed of a cloud, an application suite description and middleware services. We assume that the only difference lies in the actual applications that are running, that is, in the application suit descriptions.

If there are no applications running, then only the cloud has to be split or unified, which is trivial to do if we assume the gossip-based implementation described above. Note that the unification or splitting of the cloud is a low-level operation that has to do only with maintaining a connected random overlay network of the participating nodes.

If there are applications, then, in the case of unification, we need to unify the descriptions first and issue the new description as an update on both systems. In parallel with this, the two clouds need to be unified. This is all that needs to be done: the rest is taken care of by the self-organization mechanism provided by the middleware services. The applications will only experience an increased churn rate, or maybe a sudden increase or decrease of the number of nodes they run on, depending on how exactly the descriptions were unified. However, due to the bootstrapping service, any complex overlay structures can readily be provided even in this case.

In the case of splitting, again a lot depends on the intended function of the two resulting systems. First of all, the descriptions have to be split, followed by splitting the cloud, and then providing the new descriptors to the resulting parts. The network can be split also simply by splitting the cloud and leaving the description intact. In this case we automatically get two new systems with identical structure but fewer resources. The applications will experience a sudden decrease in the amount of resources in this latter case.

5. EXAMPLE APPLICATIONS

An application has an associated protocol, without any central components, that can run on a cloud. It will typically be implemented using a P2P approach and will be designed to run in large-scale and unreliable environments. We discuss here some possible example applications, increasing in complexity.

First of all, not all protocols running on a cloud have to be associated with only a single application; instead, some of them may be simple services extending our middleware with additional functions. For example, consider a monitoring service based on aggregation [4], that keeps administrators informed about the current system state, and possibly intervenes by appropriately modifying the application suite description (e.g., by shutting down a “secondary” application in case of a catastrophic failure of a large portion of

the cloud). As another example, consider a registry service based on, for example, a DHT, used by external entities to obtain enhanced information about the applications running in the system.

An important observation here is that such services can be implemented and deployed on the same cloud that will be used for hosting user applications. In other words, we can assume that there is no need for extra resources besides those that are already part of the cloud. Such a deployment is easier if a cloud is indeed owned by, in principle, a single organization. This same approach is followed when considering user applications.

For example, consider an instant messaging application. An ISP may dedicate a subcloud to this service, dimensioning it based on the foreseen popularity (and thus load). Later, if the application becomes more popular, the size of the subcloud may be modified adaptively. New features may be added by simply distributing new code. In contrast, in current systems updates are left to the will of users and several versions of the same application co-exist. Apart from modifying the resource allocation requirements in the application suite description, no further action needs to be taken: the system automatically deals with churn and even catastrophic failures and other sudden changes in the available resource pool.

The applications discussed so far are relatively simple and their execution is managed by a single authority through the configuration of the application suite description. However, an additional possibility is to run complex applications that can themselves contain multiple services and applications. In this case, the structure is recursive; subclouds may be organized into sub-subclouds, using another instance of the cloud middleware running inside a subcloud. Even different resource management middleware can be adopted to fulfill different requirements. The control over applications executed in subclouds may be delegated to other (potentially multiple) authorities.

To illustrate this point, consider the case of desktop grids; possible scenarios include an ISP that decides to sell part of the computing and storage power of its networked set-top boxes to paying clients, or a potentially large number of organizations that decide to federate their clouds to take advantage of the aggregated computational and informational resources available at each organization.

In such a scenario, a grid service could be run on one of the subclouds; when a complex task is submitted to the grid for execution, a suitable subset of nodes is selected, and the specific task is run over them. The selection can exploit the slicing ability of our middleware, or more complex resource management schemes can be used, e.g., based on quality-of-service agreements. Once selected, the nodes may self-organize into complex overlay topologies using the bootstrapping service and maintain these using the churn-handling service.

6. SUMMARY

In this short position paper, our goal was to outline some initial ideas that we believe could eventually lead to a middleware layer to enable very large and unreliable sets of networked computing devices to act as a platform for fully distributed applications and services, in an easily manageable manner.

We have proposed abstractions — a cloud and resource assignment based on subclouds — and we have outlined ideas towards a possible middleware implementation as well. This implementation is purely gossip-based: as a consequence, it is potentially cheap, lightweight and robust. The middleware layer is also transparent, because the mapping of the application suit description to the actual cloud and the running applications is fully automated.

Some of the protocols have been studied to some degree: for example the bootstrapping service, that allows applications to start new DHTs on demand, and to recover from catastrophic failures involving a large number of nodes, is know to be efficient and fast as well.

Clearly, until the system becomes a reality, many questions remain to be solved. Most importantly, we need to develop novel protocols for our specific resource assignment approach and various other functions, and study the possible complex interaction of the many simple components, which forms an interesting research agenda.

7. REFERENCES

[1] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In S. A. Brueckner, G. Di Marzo Serugendo, D. Hales, and F. Zambonelli, editors, *Engineering Self-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers*, volume 3910 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2006.

[2] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In H.-A. Jacobsen, editor, *Middleware 2004*, volume 3231 of *Lecture Notes in Computer Science*, pages 79–98. Springer-Verlag, 2004.

[3] 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 (P2P 2006)*, 2006. to appear.

[4] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.

[5] M. Jelasity, A. Montresor, and O. Babaoglu. The bootstrapping service. In *Proceedins of the 26th International Conference on Distributed Computing Systems: Workshops (ICDCS WORKSHOPS)*, 2006. International Workshop on Dynamic Distributed Systems (IWDDS), to appear.

[6] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P 2005)*, pages 87–94, Konstanz, Germany, August 2005. IEEE Computer Society.

[7] J. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. Epema, M. Reinders, M. van Steen, and H. Sips. Tribler: A Social-Based Peer-to-Peer System. In *The 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, February 2006.