OddCI: On-Demand Distributed Computing Infrastructure

Rostand Costa[1,2], Francisco Brasileiro[1]

[1]Federal University of Campina Grande
Systems and Computing Department
Distributed Systems Lab
Av. Aprígio Veloso, 882 - Bloco CO – Bodocongó
Campina Grande, Paraíba, Brazil +55 83 33101365

{rostand.costa, fubica}@lsd.ufcg.edu.br

Guido Lemos Filho[2], Dênio Mariz Sousa[2]

[2]Federal University of Paraíba
Informatics Department
Digital Video Applications Lab
Campus I - Cidade Universitária
João Pessoa, Paraíba, Brazil +55 83 32167093

{guido, denio}@lavid.ufpb.br

ABSTRACT

The availability of large quantities of processors is a crucial enabler of many-task computing. Voluntary computing systems have proven that it is possible to build computing platforms with millions of nodes to support the execution of embarrassingly parallel applications. These systems, however, lack the flexibility of more traditional grid infrastructures. On the other hand, flexible infrastructures currently available can gather only dozens of thousands nodes. We propose a novel architecture for generic Distributed Computing Infrastructures (DCI) that can be instantiated on demand to be, at the same time, flexible and highly-scalable. Bringing the scalability from voluntary computing, the flexibility from grid computing and the elasticity from cloud computing in a single arrangement, our proposal allows for fast setup, fast initialization and fast dismantle of customized DCI supported by both dedicated and shared underlying infrastructures. Our approach leverages broadcast communication as an efficient mechanism to enable aggregation of geographically distributed computing resources, including millions of non-traditional processing devices such as PDA, mobile phones and Digital TV receivers, using both opportunistic and non-opportunistic models. We show the feasibility of the proposed architecture by implementing it atop a digital television system. We also assess the performance of such system and show that it can be used to execute several classes of many-tasks computing applications with very high efficiency, substantially decreasing their response time.

Categories and Subject Descriptors

C.1.4 [**Parallel Architectures**]: *Distributed architectures.*

General Terms

Management, Performance.

Keywords

Distributed computing infrastructure; high-throughput computing; grid computing; cloud computing; many tasks computing; digital TV; broadcast; on-demand instantiation.

MTAGS '09 November 16th, 2009, Portland, Oregon, USA

1. INTRODUCTION

Parallel processing is a key technology to allow the timely processing of the ever increasing quantity of data that is currently being generated by sensors, scientific experiments, simulation models, and ultimately as an effect of the digitalization era that our society as a whole is experiencing. Some of the workloads that need to be processed are so large, that the only feasible way to handle them is to break the processing in a very large amount of loosely coupled sub-tasks and run them in parallel in as many processors as one possibly can. The term *many-task computing* (MTC), has recently been coined to refer to this kind of parallel processing [1].

The aggregated processing throughput achieved by scheduling as many sub-tasks as possible to run in parallel allows speeding up the execution of the application, substantially reducing its makespan[1]. In turn, large amount of parallelism can only be achieved if there is a relatively high level of independency among the sub-tasks that comprise the application and the scheduler has access to a huge number of processors. In this paper we are concerned with the latter issue, i.e. providing ways to assemble large pools of processors for the execution of MTC applications.

Desktop grid computing has proved itself as a suitable environment for high-throughput computing. Condor [2] is arguably the most well known representative of the existent technology to enable high-throughput desktop grids. Other systems that followed Condor's philosophy have also proven to be equally effective [3][4]. These generic infrastructures are, however, limited scale systems. Even if some sort of incentive mechanism is used [5], it is unlikely that a system comprising more than a few dozens of thousands of computers will ever be assembled. Indeed, the largest existing systems using these technologies feature less than a few thousands of computers [6].

Voluntary computing platforms [7][8], on the other hand, are able to assemble huge amounts of resources to process the extremely large workload of their typical applications. These powerful infrastructures are, however, less flexible in the types of applications that they support. Firstly, setting up a voluntary computing infrastructure has a cost that is significantly higher than that associated to the assembling of desktop grids; this mainly due

[1] The application's makespan is a key metric for measuring the efficiency of the execution of an MTC application; it is given by the difference between the latest completion time among all sub-tasks of the application and the submission time of the application.

to the fact that substantial effort is needed to convince volunteers to participate. Hence, they tend to be more suitable to run long-lived MTC applications whose workloads are virtually endless [7]. Moreover, the effectiveness of the gathering of resources is deeply influenced by the perceived impact of the application that is going to be executed over them. As a result, only a few applications have been able to benefit from the extremely high throughput that voluntary computing systems can deliver.

More recently, *Infrastructure as a Service* (IaaS) has appeared as a suitable technology for instantiating on-demand computing infrastructures [9]. Some companies are starting to offer the possibility of setting up systems that assemble large numbers of virtual machines, providing an interface similar to that of desktop grids [10]. This obviates the effort to gather a large number of desktops, replacing them by virtual machines running on server farms of IaaS providers. Although very flexible and simple to set up, achieving extremely high-throughput computing in these infrastructures is not straightforward, considering the available implementations.

A flexible distributed computing infrastructure able to achieve extremely high throughput would allow many MTC applications to have their response time reduced in orders of magnitude, with substantial gains for their users. Speeding up data processing in a significant way may have an unprecedented practical impact. For instance, time-to-market of products may be shortened, continuous optimization of the production line may greatly reduce production costs, knowledge extracted from the mining of huge amounts of data may improve business efficiency, cure for diseases may become a reality much sooner, etc.

Unfortunately, as discussed above, currently available technologies have fundamental shortcomings that limit either their scale or their reach. In this paper we propose a novel architecture for high-throughput distributed computing that is at the same time flexible and highly scalable, therefore serving the purpose of supporting the efficient execution of short-lived MTC applications. We provide evidences of the effectiveness of the proposed architecture by studying one possible implementation of it over a network of Digital Television (DTV) receivers.

The remainder of the paper is organized as follows. In Section 2 we present in more detail the requirements that need to be fulfilled by a system that is able to provide extremely high throughput to a range of MTC applications. We also discuss why current technologies are unable to simultaneously address all the requirements of such applications. Then, in Section 3 we describe a novel architecture that supports these requirements. We discuss, in Section 4, how this architecture can be implemented in the context of a Digital TV system. In Section 5 we assess the performance of this system. Related work is surveyed in Section 6. Finally, we present our concluding remarks in Section 7.

2. SYSTEM REQUIREMENTS

The throughput achieved when running MTC applications over a distributed computing infrastructure (DCI) depends directly on the scale it allows. In this context, the size of the processing pool is the main performance enabler, while the scheduling coordination and sub-tasks synchronization overheads are the main factors that limit performance. In order to achieve extremely high throughput, it is necessary to efficiently operate at extremely high scale. In other words, assuming that the amount of synchronization between sub-tasks does not prevent a large proportion of the workload to be executed in parallel, MTC applications can easily benefit from the availability of a massive pool of processors to increase their throughput, provided that neither the assignment of work to the available processors nor the provision of any required input as well as the collection of the output generated by the sub-tasks end up being a bottleneck.

Efficient use of the infrastructure by MTC applications requires the ability to instantiate a large pool of resources to an application whenever needed and only for the duration of the application execution. These resources can later be reassigned to different applications regardless of their task granularity and processing requirements. Moreover, to allow the execution of an unbounded number and different kinds of applications, it is essential that the infrastructure setup, including the installation of any application specific software component, can be lightweight in terms of complexity, and agile in terms of time, even taking into consideration that the target scale may be in the order of millions of processing nodes. In other words, the user must be able to easily and quickly customize the entire processing infrastructure to its needs.

In summary, in order to provide extremely high-throughput computing to a large number of applications, we envisage that a DCI must meet the following requirements:

I. *extremely high scalability*: it must be able to handle up to hundreds of millions of processing resources in the same way that it handles a few dozens of them;

II. *on-demand instantiation*: it must offer mechanisms for discovery, assemblage and coordination of the required resources, on demand and for a specified amount of time; and,

III. *efficient setup*: the configuration of the processing nodes and the system *backend* (in charge of specific application management activities such as scheduling and I/O processing) must be carried out quickly and with minimal effort, demanding no individual or specialized interventions.

In **TABLE I** we show how currently available technologies address these requirements. As it can be seen, all the requirements are addressed by at least one of the available solutions, but no technology is able to simultaneously address all requirements.

Table I - How available technologies address the requirements for flexible and extremely high-throughput computing

Requirement	Available Technologies		
	Voluntary Computing	*Desktop Grid*	*Infrastructure as a Service*
Extremely High Scalability	✓		
Efficient Setup			✓
On-demand Instantiation		✓	✓

Voluntary Computing has proven to be suitable to provide extremely high throughput. However, this can only be achieved if significant effort is devoted to convince voluntary participants to join the system which, in turn, depends in greater or lesser extent on factors such as the merit and public appeal of the application,

the amount of media coverage received, explicit advertisement campaigns in popular media, viral marketing, incentives to volunteers and other public relations activities [11]. Scalability in the deployment is achieved by making this task extremely simple and by having the resource owner actively involved in the system setup. Deployment is basically reduced to the download of a piece of software that can be easily installed by the owner of the resource. For example, a user donating her computing resources to the SETI@home [7] or FightAIDS@home [12] projects must install specific applications, each with its own protocols and parameters.

If on one side the involvement of the user allows deployment in millions of resources to be cost-effectively attained, on the other side, it makes the growth of the infrastructure slow and out of the control of the voluntary computing infrastructure provider. Moreover, changes in the software installed in the resources are harder to be accomplished, unless some automatic update procedure is provided. This, in turn, may increase security concerns from the part of the volunteers and ultimately negatively affect their willingness to join the system. Furthermore, the intrinsic singularity of each application and its need for initial setup, considerably diminishes the flexibility of these platforms. Once a resource is configured for supporting a specific voluntary computing project, it may not be shared with other similar initiatives unless explicit actions from volunteers are taken. Note that this is true even for those platforms that support multiple projects, such as BOINC [8], where the volunteer must, explicitly, attach the desired projects (or all them) to her and to determine which resources she wants to share with each project [11].

Considering desktop grid computing systems, although they provide the necessary mechanism for on-demand instantiation, their main limitations are their slow setup and relatively low scalability. The customization of the processing environment is time consuming, since each resource needs to be individually configured, whenever a change is required. The lack of scalability is mainly a result of the fact that security concerns are amplified by the increased flexibility on the kinds of applications that are supported, as well as on the increased number of application providers allowed. Since resources are spread over different administrative domains, each imposing its own security policies, it is more difficult to have a large number of resource providers agreeing on a set of compatible policies. Moreover, in peer-to-peer grids, where a reciprocation behavior is expected, there is the additional need for controls on the way grid resources are shared, such that free-riding is inhibited [5].

IaaS is also insufficient to handle all the above listed requirements. Current implementations allow only a few virtual machines to be automatically instantiated. For larger systems, off-line negotiation is required. Even then, it is unlikely that any current provider will be able to accept the instantiation of a system with millions of virtual machines for the short period of time required to run a typical MTC application. More recently some of the largest IaaS providers are offering the possibility of reserving virtual machines for future use. However, reservations need to be made for periods of at least one year. Finally, efficient setup of the application environment takes advantage of the existence of a companion storage service that allows fast access to data. Virtual machine images can be customized, stored in these systems and later used to create the required environment for the virtual machines that are instantiated. If millions of virtual machines could be instantaneously instantiated, concurrent access to the shared storage by millions of clients would certainly produce a bottleneck on the storage server.

In the following section, we present a novel architecture which is able to address all the requirements for flexible and extremely high-throughput computing.

3. NOVEL ARCHITECTURE

In order to build a DCI capable of meeting the requirements identified in the previous section, especially those related to scalability, it is necessary to provide a way to access a potentially enormous quantity of processors, send to all of them programs and possibly data, remotely trigger the execution of the code staged, gather the results produced, and finally release the allocated resources so that other applications can use them. As discussed before, this is definitely not supported by the current approaches based on clusters, server farms, desktops in the Internet edges, or traditional grids. Therefore, it is necessary to expand the alternatives beyond the boundaries of traditional corporate and personal desktop computer networks.

The question is then, where to find dozens of millions of available processors and to configure them accordingly and instantaneously for the use of MTC applications? Moreover, how to perform this formidable task with minimum delay? Fortunately, there are new possibilities brought by the emergence of new services and devices that combine technologies developed for different contexts and scenarios, such as mobile phones, digital TV receivers (a.k.a. set-top boxes), PDAs and practically every digital gadget connected to the Internet. All these devices are equipped with reasonably powerful processors and fairly large memory, allowing them to support the execution of applications. Indeed, there is a myriad of digital devices, computationally capable, virtually connected and possibly underutilized that, if properly coordinated, may represent an unprecedented potential for the processing of MTC applications. The current trend for connectivity-oriented devices pushes the industry towards scenarios of convergence and helps to reinforce the idea of exploring these new powerful devices.

In this work, we consider a special category of such devices, namely those which may be organized as a *broadcast network*. A broadcast network has the potential to access simultaneously all the connected devices which can be coordinated to accomplish some action. By transmitting a piece of software through the broadcast channel to be loaded simultaneously and indistinctly by all processors embedded in the devices connected at a given time, it is possible to build, in a very fast[2] and controlled way, a DCI, capable of running distributed applications in general, and MTC applications in particular.

3.1. On-Demand Distributed Computing Infrastructure

In this subsection we present a novel architecture for generic DCI that is at the same time flexible and highly scalable. A noticeable characteristic of this architecture is that, differently from any other alternative for distributed computing, no previous identification

[2] In fact, how fast the software will be uploaded depends on the size of the software footprint and on the capacity of the broadcast channel. In Section 5 we discuss this issue in more detail.

and registering procedures are, necessarily, required for the existing resources. Put simply, the DCI does not exist until it is requested and activated through the broadcast channel. Due to this singularity, we call the architecture *On-demand Distributed Computing Infrastructure* (OddCI) and the activation of one instance of an OddCI as the *wakeup process*.

In the proposed architecture, a standard broadcast network is augmented by four components, namely a *Provider, a Controller*, a *Backend* and *Processing Node Agents. M*oreover, we assume that processing nodes are accessible by the broadcast network and are also able to communicate with both the *Controller* and the *Backend* via individual full-duplex one-to-one communication channels, called the *direct channels* (see Figure 1).

The *Provider* is responsible for creating, managing and destroying the instances of OddCI according to the user´s requests. The *Provider* sends the suitable instructions so that the required OddCI instance can be dynamically provisioned or released by the *Controller*.

The *Controller* is in charge of setting up the infrastructure, as instructed by the *Provider*, by formatting and sending through the broadcast channel the control messages, including software images, necessary for building and maintaining the OddCI instances. It receives information from the active processing nodes about their status and configuration and consolidates and passes this information to the *Provider*.

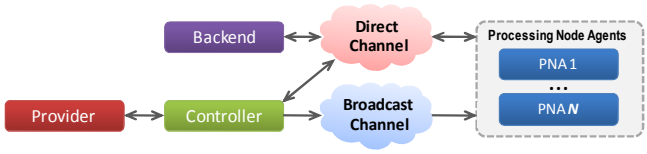

Figure 1 - OddCI Architecture Overview

The *Backend*, in its turn, is responsible for managing particular activities of each running application. These activities may include scheduling, provision of input data, as well as gathering and, possibly, post-processing of the output generated by the parallel application.

The *Processing Node Agents (PNA)* running at each processing unit accessible by the broadcast network, are responsible for listening the broadcast channel and process the control messages sent by the Controller, managing the load of new images in the memory of the processing node and the execution of the loaded image. A PNA is structured as illustrated in Figure 2.

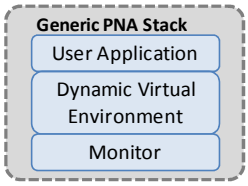

Figure 2 - Processing Node Agent Stack

The *Monitor* interacts with the *Controller* through the broadcast channel, listening and processing the *control messages*, loading new images in the *Dynamic Virtual Environment* (DVE) [13] and managing the execution of the loaded image. The *Monitor* communicates with the *Controller* through the direct channel for reporting its current status. The DVE enables a safe and suitable space to execute the *User Application*, safeguarding the interests of the owner of the device, of the *User* and of the operator of the broadcast network, if any. Finally, the *User Application* is the loaded image and performs the specific processing wanted.

3.2. OddCI Operation

The *User* submits a request for a DCI to the *Provider* indicating the requirements for the resources and providing a specific application image, including programs, common data and the size of the DCI required. The *Provide*r analyses the order and forwards *a wakeup request* to the *Controller* to allocate the requested resources and create an OddCI instance.

Upon receiving such request, the *Controller* triggers the *wakeup process*, formatting and transmitting the appropriate *control message* through the broadcast channel to the processing nodes.

The PNA are configured to only accept messages broadcast by their associated *Controller* (this can be easily achieved through a digital signature mechanism). When a processing node is switched on, the PNA initiates its execution and starts sending heartbeat messages to the *Controller* through the direct channel. These messages contain the PNA's state and the identification of the OddCI instance to which it currently belongs (if any).

An active PNA can be either idle or busy, executing the software image of a particular instance of an OddCI. Upon receiving a wakeup message, if the PNA is not idle, the message is simply dropped; otherwise, the PNA assesses its own compliance with the requirements present in the message and, if there is a match, it creates a DVE for loading and executing the user's application present in the message received. Then, the PNA's state switches from idle to busy. Idle PNA may also drop wakeup messages. This is controlled by a parameter contained in the wakeup message that specifies the probability with which the message should be handled by an idle PNA. The *Controller* may also broadcast reset messages to destroy an OddCI instance, or adjust OddCI exceeding size replying *heartbeat messages* with a reset command, through the direct channel, to a particular PNA. The busy PNA composing the specific OddCI instance handle the message, destroy the DVE and change their status to idle. These features allow the *Provider* to command the creation, dismantle and resizing of several OddCI with different sizes.

All PNA periodically send *heartbeat messages* to the *Controller*, informing it about the PNA's current status. These messages contain the PNA's state and the identification of the OddCI instance to which it currently belongs (if any). With this information the *Controller* is able to take appropriate management decisions when instantiating an OddCI for a new application that has been submitted, and to keep the active OddCI instances at their appropriate size. Since millions of PNA may be simultaneously sending heartbeat messages to the *Controller,* the PNA must be appropriately configured by the *Controller* so that the handling of these messages will not consume too much of the *Controller's* processing and networking resources[3]. The *Controller* consolidates all the status information received from the PNA and passes it to the *Provide*r.

[3] A discussion on possible mechanisms that avoid the Controler from becoming a bottleneck is out of the scope of this paper and it will be theme of our future research.

Since a PNA can generally be switched off at the will of its owner, from time to time the *Controller* may need to retransmit *wakeup control messages* to recompose OddCI instances that have lost some of its PNA.

3.3. OddCI Enabling Technologies

Nowadays, several technologies can already be used to make possible the simultaneous and unidirectional communication among digital devices in the model of one-to-many, characteristic of the concept of broadcast network evoked in this work. Besides the traditional diffusion of TV, in their new digital version and in their different modalities (satellite, terrestrial, cable, mobile, etc) [14], we can also mention the multicast transmission by broadband networks , BitTorrent, mobile phone networks, and video transmission (VoD, WebTV, IPTV etc). By taking advantage directly of functionalities made available by native resources for such technologies or by doing some complements or adaptations, it is possible to build implementations of OddCI for several contexts.

In the same way, it is also quite wide the diversity of devices that can be reached by one or more of the mentioned broadcast technologies, from computers to equipments with more specific purposes such as game consoles, mobile phones and digital TV receivers. Some of these less traditional devices already proved their potential use for distributed processing in voluntary computing projects [15][16].

4. ODDCI OVER A DIGITAL TV NETWORK

In this section we discuss how the architecture presented in the previous section can be implemented with currently available technology. Among the many possibilities of technologies and applicable devices for a first implementation of the architecture proposed for OddCI, we opted for the choice of Digital TV Networks for the following main reasons:

- It is a technology open with well-defined standards and that supports native transmission of data in broadcast;
- It is in fast global expansion, being implanted at the moment in many countries with deadlines for deactivation of analogical TV [17];
- Great spectrum of devices addressed: ranging from the traditional set-top-boxes to mobile devices;
- Feasibility of constructing a testbed, when compared to alternatives such as cell networks and game consoles;
- Our previous experience with Digital TV and Grid Computing [18].

We name this implementation OddCI-DTV. Next, we provide some basic background on DTV systems, followed by the description of the proposed implementation.

4.1. Background

A graphical representation of a Digital Television (DTV) network can be seen on Figure 3.

The DTV receiver can be seen as a computer adapted for the needs of the television environment, having several processors – one of them dedicated to run interactive applications –, memory, non-volatile storage device, network adapter, operating system etc. It also runs a middleware, which is responsible for abstracting specific hardware characteristics of each receiver, allowing the same application to be executed on set-top boxes made by different manufactures.

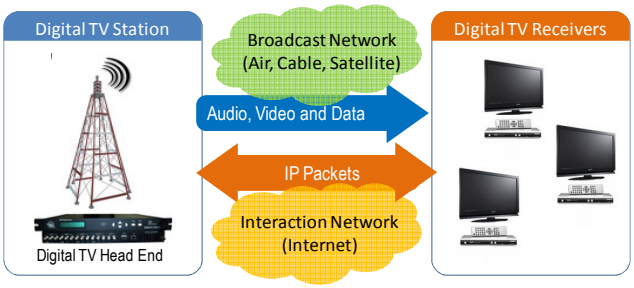

Figure 3 - A common structure for a DTV network

Most of the middleware available nowadays, such as the DVB-MHP [14][19], the ATSC-ACAP [14] and the Brazilian SBTVD-Ginga [20] chooses Java as part of the solution for the execution of applications on the receivers. The Java applications executed on the receivers are called *Xlets* [21].

It is also important to notice that not all TV programs use the interactivity features of the DTV system, and many only barely use them, consuming only a tiny amount of the available resources (excess bandwidth in the broadcast channel and dedicated interactive applications' processor). In fact, due to the nature of many programs broadcasted, it is very likely that these resources will never be used at their full capacity.

In a DTV system, application and data are multiplexed together with audio and video streams. This process follows the Digital Storage Media Command and Control (DSM-CC) specification [22]. DSM-CC supports the broadcasting of a file system using the object carousel mechanism [19], which allows large volumes of data to be transmitted to a set of receivers, cyclically repeating the transmission of its content in modular units, known as object carousel. Data are cyclically repeated to allow those receivers that are being switched on in the middle of transmission or those having slight different processing capacity to have access to the data at different times. If an application on the receiver wishes to access a certain file of the object carousel, the access is delayed until the next data retransmission for that particular file. It is possible to dynamically update the carousel that is being transmitted, adding, removing or altering its files, only by creating a new version of the module carrying the files to be updated at any point in time. The generated data flow is then multiplexed together with all other elementary flows such as audio and video flows. Most systems adopt the ISO/IEC 13818 (MPEG-2) standard [14]. All flows then are mixed together in a transport stream and then transmitted to the DTV receivers.

4.2. Running Applications on the DTV receiver

Using the abstraction of a file system supplied by the object carousel, the applications and their data are continuously transmitted, multiplexed with audio and video and additional control information (meta-data). This information is demultiplexed at the receiver and adequately handled by the middleware and other components.

To signal a receiver that applications are available, the transport stream includes additional information named Application Information Table (AIT) [14][22]. The AIT contains all the information the receiver needs to run the application, such as the name, the identifier and the control of the application lifecycle.

The latter is signaled by the AIT field *application_control_code*, which allows the broadcaster to signal the receiver what to do with the application regarding its initialization. When *application_control_code* indicates the value `AUTOSTART`, the application must start immediately without the user intervention; these applications are named *trigger* applications.

In a DTV receiver, several applications may be running at the same time, and there is a need to enforce some separation between the applications. Xlets are a concept similar to *applets* [23]. They have been introduced by Sun in the JavaTV specification and adopted as the Java application format for most DTV middleware standards.

Once the Xlet arrives at the DTV receiver, the application manager, a component of the DTV middleware, which controls the possible Xlet states, handles it. The Xlet states are *Loaded*, *Paused*, *Started*, and *Destroyed* [14][23][24]. The complete lifecycle of an Xlet is depicted in Figure 4.

The application manager loads the Xlet's main class file and creates an instance of the Xlet by calling the default constructor. Once this has happened, the Xlet is in the *Loaded* state. When the user chooses to start the Xlet (or the AIT indicates that the Xlet should start automatically), the application manager calls the `initXlet()` method, enabling the Xlet to initialize itself, possibly loading any additional data from the object carousel. When the initialization is complete, the Xlet is in the *Paused* state and is ready to start its execution. When the application manager calls the `startXlet()` method, the Xlet is moved into the *Started* state, and is now able to interact with the user (if it is programmed to do so). During the execution of the Xlet, the application manager may call the `pauseXlet()` and `startXlet()` methods several times, moving the application to *Paused* state and *Started* state, respectively. At the end of the Xlet's life time, the application manager calls the `destroyXlet()` method, which causes the Xlet to go to the *Destroyed* state, freeing all its resources. After this point, this instance of the Xlet can no longer be started [24].

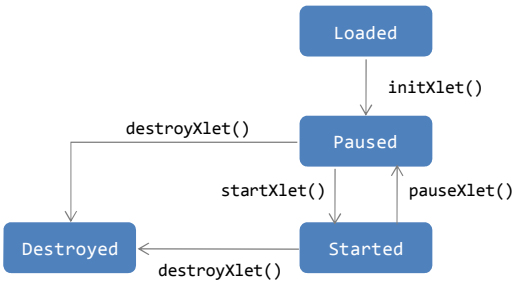

Figure 4 - The state diagram for an Xlet

Security and reliability are key considerations in Digital TV, and the standards of the segment have a number of security measures to ensure that applications do not affect the middleware [14]. For example, the receiver can authenticate downloaded applications signed by application developers or transmitters and the user and network operators can set limits on what an application can do.

4.3. OddCI-DTV Implementation

In order to deploy the OddCI architecture presented in Section 3 over a DTV network, we need to implement the four specific software components discussed earlier, namely: the *Provider*, the *Controller*, the *Backend* and the *PNA*. In Figure 5 we highlight the currently available technologies for the DTV segment that can be used and how they are associated with the elements of the generic OddCI architecture.

In an OddCI-DTV, the *wakeup process* is implemented by coding the PNA as an Xlet application that is configured with the *application_control_code* set to the value `AUTOSTART`. When an OddCI instance creation is commanded by the *Provider*, the *Controller* configures the carousel to transmit a *control message* composed by the PNA Xlet and two other files: the "*image*" file, that contains the image of the user application, and the *configuration* file that contains a number of attributes. The algorithm of the PNA Xlet is an infinite loop that sends a heartbeat message to the *Controller*, possibly executes some action based on the message received, and finally sleeps for the time indicated in the configuration file, before starting a new iteration of the loop.

Note that since the PNA Xlet is a trigger application, when the new carousel starts to be broadcasted, every set-top box that is tuned in the correspondent channel will load the PNA Xlet and start to execute it.

If the state of the PNA Xlet is idle, and the information in the "*message_type*" attribute of the configuration file indicates that it is a *wakeup message* that is being broadcasted, then the PNA Xlet reads the "*probability*" attribute and, with the probability indicated in this file decides whether it should load and execute the application contained in the file *image*. If the application is to be executed, first its identification is read and saved in the PNA Xlet memory. The PNA Xlet sets its state to busy, creates a new process to run the DVE and loads the user application inside. If, on the other hand, the message type is "*reset*", then the PNA does nothing.

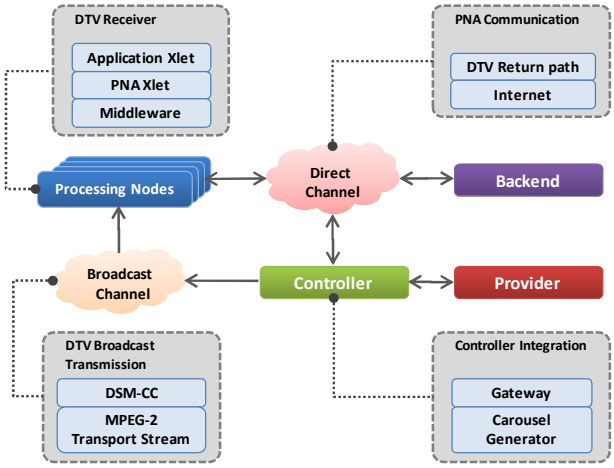

Figure 5 - Current DTV technologies able to support the OddCI-DTV implementation

When the PNA Xlet is in the busy state and the message is a "*reset*" message, the PNA reads the "*oddci_id*" attribute and if it matches the identification of the OddCI instance to which the PNA currently belongs, then it kills the process that is running the DVE and sets its state to idle. Similarly, the PNA Xlet will not perform any action if it is busy and the message type is "*wakeup*".

In the above discussion we considered that the trigger application is broadcasted in a particular TV channel. In some cases, the Controller can request that the TV station transmit the same application in different TV channels at the same time. Having

multiple channels to distribute the trigger application (PNA Xlet) increases the potential number of receivers connected with a direct impact on the maximum size of the OddCI-DTV systems that can be instantiated.

4.4. Proof of Concept

To demonstrate the feasibility of the OddCI-DTV architecture, we have built a Java prototype. Besides a Provider, a specific Controller was developed able to interact with the data carousel and to inject *control messages* (including the PNA XLet, application images and configuration data). The *wakeup process* triggers a generic PNA Xlet, with possibility of dynamic adaptation for several environments of DTV, like DVB-MHP [23] or SBTVD-Ginga [20]. The prototype worked perfectly in two emulators of DTV middlewares, XletView [25] and OpenGinga [26]. The parallel application example used is composed of a backend module based on the paradigm of voluntary computing and a Java client module that implements the interface *Runnable.* The client module did not need to be an Xlet because the PNA Xlet acts as an adapter concentrating the communication with the set-top box (STB) middleware.

Besides the execution of the prototype using STB emulators, we also made some micro benchmarking using a real STB. The accomplished tests had two main objectives: a) compare the performance of an STB with a standard desktop; and b) to evaluate the processing performance loss when executing the application with the STB in standby and in normal use.

The selected application for the tests was BLAST (Basic Local Alignment Search Tool) [27], a bioinformatics algorithm for comparing primary biological sequence information, such as the amino-acid sequences of different proteins or the nucleotides of DNA sequences. A BLAST search compares a query sequence with a library or database of sequences, and identifies library sequences that resemble the query sequence above a certain threshold. It is available for download at the U.S. National Center for Biotechnology Information (NCBI) website [28].

Using a cross compiler, we ported the NCBI Toolkit for an STB based on STI Microeletronics's processor ST7109 [29] with 32MB of memory flash and 256 MB of RAM. Representing varied workloads, the tests were accomplished using the BLASTALL and BLASTCL3 programs and they were divided into three categories: local processing with small databases (#1-9), local processing with large databases (#10-12) and remote processing (#13-15). A total of 15 experiments were made in the STB in both "use mode", with a TV channel tuned, and "standby mode", with the middleware in an inactive state. The same tests are reproduced in an x86 Linux in a reference PC (Pentium Dual Core, 1.6 MHz, 1Gb RAM with Debian Linux). The results are showed in Table II and Table III.

To accomplish objective a), we used the program BLASTALL with different input parameters. We computed the average performance decrease for the samples presented in **TABLE II** with a confidence level of 90%. The average performance of the STB, when compared to the PC, was 20.6 worse with a maximum error of 10%. Nevertheless, the results also show that the largest workload (test #12) spent almost 11 hours to be concluded in an STB in normal use. Considering that a work unit of the project Folding@Home for use in PS3 is projected to run for 8 hours [15], an efficient use of DTV receivers can be obtained with an appropriate relationship of granularity of the tasks versus the amount of available nodes. The results also show that the average performance reduction when comparing the execution times for the STB in standby and in normal use is 1.65, with a maximum error of 17%.

Table II - Processing time obtained in the execution of Blastall program in STB and PC.

#Test	DTV Receiver		PC with
	In Use (s)	*Standby (s)*	x86 Linux (s)
1	3.338	1.356	0.556
2	2.102	1.333	0.041
3	5.185	3.208	0.076
4	0.179	0.117	0.015
5	0.173	0.116	0.016
6	0.175	0.116	0.013
7	1.026	0.612	0.293
8	0.944	0.610	0.023
9	1.642	0.090	0.025
10	0.177	0.118	0.015
11	9314.247	6315.410	213.770
12	38858.298	26973.262	747.372

In order to check the capacity of an STB to communicate appropriately with a *Backend* using the direct channel for obtaining tasks and to send results, we used the BLASTCL3 program. This program submits a sequence to be looked for in databases of NCBI, receives the result and writes it in a file. As the search processing is run remotely, the most relevant aspect in this experiment is the way how the STB handles data over the network connections. In this case, as it can be verified in **TABLE III**, there is no significant performance difference between the PC and the STB. Eventual NCBI server load or network traffic can explain the test #13, in which STB took less time than PC.

Table III - Processing time obtained in the execution of Blastcl3 program in STB and PC.

#Test	DTV Receiver		PC with
	In Use (s)	*Standby (s)*	x86 Linux (s)
13	79.285	77.389	114.240
14	84.916	89.880	82.158
15	449.189	436.174	445.050

5. PERFORMANCE ASSESSMENT

In this section we assess the performance of the OddCI-DTV system. To facilitate the analysis, we assume the simplest class of MTC applications, in which all sub-tasks are independent from each other. We start by studying the overhead of the wakeup process. Then, we discuss how different applications would perform when executing over a typical OddCI-DTV system.

5.1. Overhead of the Wakeup Process

We use a simple analytical model to estimate the time required to instantiate a new OddCI-DTV instance to run a particular application. Let β be the unused capacity of the broadcast channel of the OddCI-DTV system and I the size in bits of the image that is used by the OddCI-DTV instance. Thus, considering that the bulk of the *wakeup process* is the transmission of the image to the processing nodes and that the sizes of the other files contained in the carousel are negligible when compared to I, the average time taken in the wakeup process is given by:

$$\overline{W} = 1.5\frac{I}{\beta}$$

In the best case, when the PNA Xlet starts to read the image file it does so with no delay. On the other hand, in the worst case, the PNA Xlet starts to read the image file just after it has started to be transmitted, thus, it will have to wait a full cycle of the carousel before it can start reading the file. Thus, in average it has to wait half a cycle of the carousel to start reading the file, plus a full cycle of the carousel to read the file.

Typical applications have images with a size not larger than a few Mbytes. For instance, the OurGrid Worker [3] and both versions of BLAST used in the previous section have images smaller than 8Mbytes. Also, in current DTV systems, β should be at least 1 Mbps. Thus, the delay for the wakeup process should be not larger than a few minutes. The wakeup time for setting up OddCI-DTV instances to run any of these applications in millions of processing nodes, considering β = 1Mbps, would be less than 64 seconds.

5.2. Performance of Common Applications

5.2.1. System Model and Performance Metrics

An OddCI-DTV system comprises a *Controller* with access to a broadcast channel with an unused capacity of β bps, a large number of set-top boxes, and individual point-to-point full-duplex channels with a capacity of δ bps, linking each set-top box to both the *Controller* and the *Backend* components. For simplicity, we assume that the set-top boxes are homogeneous and have all the same processing and communication capacities that we refer to as the computing and communication power of a *reference set-top box*.

Let an OddCI-DTV application, or a job for short, be a tuple $J=(I, n, T, R)$, where I is the size of job image in bits, n is the number of tasks in the job, T, $T=\{t_1, t_2, \ldots, t_n\}$, is a set of tasks that need to be fetched for execution by the application and R, $R=\{r_1, r_2, \ldots, r_n\}$, is the set of results produced by each task in T. Each task t in T is defined by a tuple $t=(s, p)$, where $t.s$ is its size in bits and $t.p$ is the time required for t to be processed on a reference set-top box. Note that for a parametric application, tasks do not need to be fetched, so, $t.s = 0$, for all tasks. Finally, each r_i in R represents the size in bits of the result's data. We consider that the *Backend* is suitably provisioned, so that the time required to process the results is negligible.

Given the above, we can calculate the execution time (makespan) of the jobs that are submitted to an OddCI-DTV instance. Let N be the number of set-top boxes that are tuned in a channel that is broadcasting a given application and that will remain tuned for at least the time required to complete the execution of the application. Let also, s, p and r be, respectively, the average size of the input data, the average processing time of the tasks, and the average size of the results. Then, the average makespan of a job J is given by:

$$\overline{M} = 1.5\frac{I}{\beta} + \left\lceil \frac{n}{N} \right\rceil \left(\frac{s+r}{\delta} + p \right) \quad (1)$$

This makespan corresponds to the minimal time to instantiate the OddCI-DTV ($1.5I/\beta$) plus the time to fetch the tasks, process them, send the results, and have them stored at the backend.

5.2.2. Analysis

In our analysis we consider the efficiency of an OddCI-DTV instance. Suppose that the application was to be executed on a single machine with the computing power of a reference set-top box. Then, the average throughput achieved by the single machine is given by:

$$\overline{T}_{single} = \frac{1}{p}$$

Now, provided that $n \geq N$, an ideal infrastructure would sustain the same throughput as the number of processing nodes in the system increases. That is to say:

$$\overline{T}_{ideal} = \frac{N}{p}$$

Thus, we define the efficiency of an OddCI-DTV instance as the ratio between the average throughput it yields ($n/\overline{M}$) and the ideal average throughput. Formally:

$$E = \frac{np}{\overline{M}N} \quad (2)$$

To gauge the efficiency of an OddCI-DTV instance we use typical values for some of the parameters of equations (1) and (2). For instance, it is safe to assume that β is at least 1 Mbps in current DTV systems. Also, 150 Kbps is a reasonable lower bound for δ, considering typical ASDL connections. Also, we set I=10 Mbytes as an average size of an MTC application. Let Φ, $\Phi=(s+r)/(\delta p)$, be the suitability of an application to execute in an OddCI-DTV. The lower is the value of Φ the less suitable is the application for execution in an OddCI-DTV. Figure 6 shows the plots for the efficiency of such an OddCI-DTV system as a function of Φ, for different values of n/N, with $(s+r)$ set to 1 Kbyte.

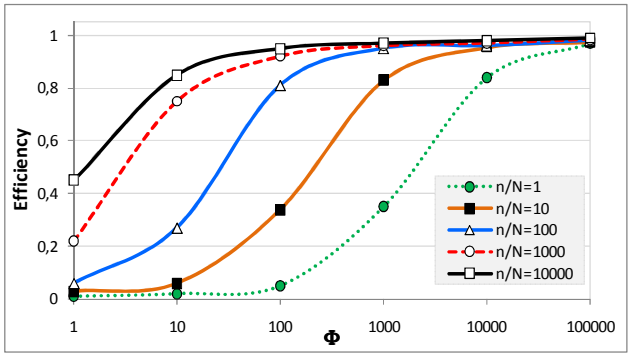

Figure 6 - Efficiency of an OddCI-DTV instance with (s+r)=1K bytes for different application classes

For the example illustrated in Figure 6, the average execution time of a task varies from 53 ms (Φ=1) to approximately one and a half hour (Φ=100,000). As it can be seen, as the suitability of the application grows, so does the efficiency of the system. Also, even for applications with very low suitability, an appropriate selection of the ratio n/N may be enough to allow reasonable efficiency values. A ratio above 100 is generally enough to yield very high efficiency for most practical applications.

Surely, efficiency is not the only metric that needs to be analyzed. From the application point of view, the makespan achieved is a much more important issue. Figure 7 shows the values for the makespan considering the same scenario discussed above (notice the log scale in the y axis).

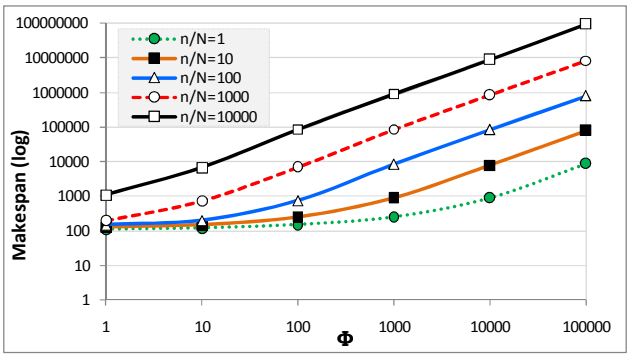

Figure 7 - Makespan of an OddCI-DTV instance with (s+r)=1K bytes for different application classes

As it can be seen, achieving high efficiency may come with a severe penalty on the makespan of the application. Nevertheless, it is always possible to find out a compromise between efficiency and performance.

Note that, differently from other infrastructures, in an OddCI-DTV system, it is always possible to precisely define the size of the instance that will run a given application. Thus, setting a suitable ratio between the number of tasks and the number of processing nodes is easily achieved.

6. RELATED WORK

To the best of our knowledge, we are the first group to investigate the potential of the broadcast communication to build instantaneous and on-demand distributed computing infrastructures. There exist, however, some works that present a certain convergence with our research.

The Falkon framework (Fast and Light-weight tasK executiON) [30][31] focuses on enabling the rapid execution of MTC applications on compute clusters based in the integration of multi-level scheduling and streamlined dispatchers to deliver high performance. The Falcon's multi-level scheduling separates the resource acquisition (by requests to batch schedulers, for example) from task dispatch, in a process similar to the OddCI approach.

The RESERVOIR Project [32] presents an architecture that allows providers of cloud infrastructure to dynamically partner with each other to create a virtually infinite pool of resources. Its model for Federated Cloud Computing is based in the separation between the functional roles of service providers and infrastructure providers, where the latter can lease resources dynamically and transparently to the former. The OddCI architecture can be applicable in this context.

Snowflock [33] is, in turn, an implementation for a Virtual Machine (VM) fork abstraction which instantaneously clones VMs into multiple replicas running on different hosts, based on a one-to-many communication scheme, like OddCI. Using a multicast distribution technique, Snowflock provides sub-second memory cloning of the active VMs that potentially can scale to hundreds of workers, consuming a few cloud I/O resources.

Considering the use of non-conventional devices for building an infrastructure to run MTC applications, we can highlight three systems. The TVGrid system [18], our preliminary work that led to the research described in this paper, the BOINCOID Project [16], an Android port of the BOINC Platform, and the grid of gaming consoles that forms the infrastructure of the voluntary computing Folding@home project [34]. Folding@home is a distributed computing project designed to carry out molecular simulations to understand protein folding, misfolding, and related diseases. Folding@home has no powerful supercomputers available for processing. Instead, the main collaborators are thousands of personal computers running a small program (client). Starting in 2006, Folding@home began to use the idle time of gaming consoles connected to the Internet to achieve performance on the PetaFLOP scale [35]. This reinforces the trend of using emergent digital devices and shows the high scalability that these devices can deliver.

7. CONCLUSIONS

We have discussed the limitations that the currently available computational infrastructures such as desktop grid, voluntary computing and IaaS have to handle generic MTC applications. We argue that extremely high scalability, efficient setup and on-demand instantiation are fundamental requirements that are not completely addressed by any of the current approaches. We presented a novel architecture, named On-demand Distributed Computing Infrastructure (OddCI), which is capable of fulfilling these requirements by exploring the new possibilities brought by the emergence of new services and devices, such as mobile phones and digital TV receivers, which can be organized as a broadcast network.

We have discussed in detail how an OddCI system can be instantiated over a traditional Digital TV network (OddCI-DTV). We have shown how the software needed can be developed on top of the existing technologies used for DTV, in particular the middleware technologies that follow the standards established for DTV.

The performance evaluation of the OddCI-DTV system considered the overhead of the wakeup process, the efficiency of the system, and the performance that it can yield to different classes of MTC applications. It was shown that as the suitability of the application grows, so does the efficiency of the system. Moreover, even for applications with very low suitability, an appropriate selection of parameters may be enough to yield very high efficiency for most practical applications.

8. ACKNOWLEDGEMENTS

Francisco Brasileiro is a CNPq/Brazil researcher (grant 309033/2007-1).

9. REFERENCES

[1] I. Raicu, I. Foster. "Many-Task Computing for Grids and Supercomputers," IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08), 2008.

[2] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," Proc. 8th Int'l Conf. Dist. Computing Systems, 1988.

[3] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray, "Labs of the world, unite!!!," Journal of Grid Computing. vol. 4(3), pp. 225—246, 2006.

[4] L. Oliveira, L. Lopes, and F. Silva, "P3 (Parallel Peer to Peer): An Internet Parallel Programming Environment." Web Engineering and Peer-to-Peer Computing Workshop. Lecture Notes in Computer Science. vol. 2790, pp. 274--288. Pisa, Italy, Springer, May 2002.

[5] N. Andrade, F. Brasileiro, W. Cirne, and M. Mowbray, "Automatic grid assembly by promoting collaboration in peer-to-peer grids." Journal of Parallel and Distributed Computing. vol. 67(8), pp. 957—966, 2007.

[6] D. Thain, T. Tannenbaum, and M. Livny, “How to Measure a Large Open Source Distributed System, Concurrency and Computation: Practice and Experience.” vol. 8(15), December 2006.

[7] D. P. Anderson, et al, “SETI@Home An Experiment in Public-Resource Computing,” Communications of the ACM Archive, vol. 45(11), pp. 56—61, November 2002.

[8] D. P. Anderson, “BOINC: A System for Public-Resource Computing and Storage,”. Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04), pp. 4-10, 2004.

[9] L. Wang, G. Von Laszewski, M. Kunze, and J. Tao, “Cloud computing: A Perspective study,” Proceedings of the Grid Computing Environments (GCE) workshop. Held at the Austin Civic Center: Austin, Texas, November 2008.

[10] Amazon Elastic Compute Cloud (Amazon EC2), http://aws.amazon.com/ec2/

[11] P. Anderson, and G. Fedak, G. “The Computational and Storage Potential of Volunteer Computing,” Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06). pp. 73--80. Singapore, May 2006.

[12] FightAIDS@home, http://fightaidsathome.scripps.edu

[13] K. Keahey, K. Doering, and I. Foster, “From Sandbox to Playground: Dynamic Virtual Environments in the Grid,” in 5th International Workshop in Grid Computing. 2004.

[14] S. Morris, and A. Smith-Chaigneau, “Interactive TV Standards – A Guide to MHP, OCAP and JavaTV,” ISBN-13 978-0-240-80666-2. Focal Press , Elsevier, 2005.

[15] Folding@home PS3 FAQ, http://folding.stanford.edu/English/FAQ-PS3

[16] Boincoid - An Android Port of the Boinc Platform, http://boincoid.sourceforge.net/

[17] Final signoff for EUA analog TV service, http://blog.taragana.com/e/2009/06/12/snow-in-the-forecast-analog-tv-broadcasts-ending-friday-prompting-consumer-confusion-9002/, June 2009.

[18] C.E.C.F. Batista, et al., “TVGrid: A Grid Architecture to use the idle resources on a Digital TV network,” Proc. 7th IEEE International Symposium on Cluster Computing and the Grid (The Latin America Grid Workshop), pp. 823--828. Rio de Janeiro, Brazil, May 2007.

[19] ETSI TR 101 202 V1.2.1. Digital Video Broadcasting (DVB), Implementation guidelines for Data Broadcasting. 01-2003. Technical Report.

[20] G. L. de Souza Filho, L.E.C. Leite, C. E. C. F Batista, “Ginga-J: The Procedural Middleware for the Brazilian Digital TV System,” Journal of the Brazilian Computer Society (ISSN: 0104-6500). vol. 4(13), pp. 47—56, 2007.

[21] Sun JavaTV - “Java Technology in Digital TV,” http://java.sun.com/products/javatv

[22] ISO/IEC TR 13818-6. Information technology — Generic coding of moving pictures and associated audio information — Part 6: Extensions for DSM-CC, 1998.

[23] ETSI Standard. TS 102 819: Globally Executable MHP (GEM), http://webapp.etsi.org/workprogram/Report_WorkItem.asp?WKI_ID=19737

[24] The Interactive TV Web: The Java TV Tutorial, http://www.interactivetvweb.org/tutorials/javatv

[25] XletView - MHP Emulator, http://www.xletview.org

[26] OpenGinga - Ginga Emulator, http://www.openginga.org

[27] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, "Basic local alignment search tool," J Mol Biol 215 (3): 403–410. doi:10.1006/jmbi.1990.9999. PMID 2231712. http://www-math.mit.edu/~lippert/18.417/papers/altschuletal1990.pdf, 1990.

[28] NCBI Blast, http://blast.ncbi.nlm.nih.gov/Blast.cgi

[29] STI Microeletronics, http://www.st.com

[30] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, “Falkon: A Fast and Lightweight Task Execution Framework,” IEEE/ACM SC, 2007.

[31] I. Raicu, Z. Zhang, M. Wilde, I. Foster, P. Beckman, K. Iskra, and B. Clifford. “Toward Loosely Coupled Programming on Petascale Systems,” IEEE/ACM Supercomputing, 2008.

[32] The Reservoir Project, http://www.reservoir-fp7.eu/

[33] SnowFlock – Swift Vm Cloning for Cloud Computing, http://sysweb.cs.toronto.edu/snowflock

[34] Folding@home Distributed Computing, http://folding.stanford.edu

[35] Folding@home Petaflop Barrier Crossed, http://blog.us.playstation.com/2007/09/19/foldinghome-petaflop-barrier-crossed