Extreme Computing Coursework 1

s0840449

November 5, 2011

1 Introduction

Dropcycle is a start-up that aims to provide users with computation as a service, allowing users to have their processes moved to the cloud when they need more resources than the user can provide. Achieving this goal while keeping costs low and potential scalability high requires an examination of the typical users and use cases for the system, an examination of issues that may arise when creating Dropcycle and potential solutions for them, and a justified decision on what choice to make for the issues encountered.

2 Design Considerations & Alternatives

Current methods for computation fall into two groups. Small-scale computation is performed on personal machines with a handful of computing resources, while large-scale computation is performed across multiple machines, such as over a computer cluster or on Amazon EC2[1]. Dropcycle is targeting the group that lies in between these two - users in need of a more powerful machine than they can afford, but do not need or cannot use full scale distributed computing. Examples of this group may include researchers working in fields for which distributed computing is excessive or not financially viable. An important feature of these groups is that they are often likely to be running software that they did not write and therefore cannot change. This means that Dropcycle must attempt to provide as inconspicuous a service as possible, able to migrate and run processes without altering them.

The technical problems faced when designing Dropcycle can be split into four categories: Process Migration & Hosting, File-System Issues, Client-Side Issues, and Communication. For each of these I will explain the issues involved and possible solutions for them.

2.1 Process Migration & Hosting

One of the main requirements for Dropcycle is the ability to transfer a running process from the client to the server, with as little interruption to functionality as possible. The target server must be able to execute client processes regardless of their origin, and must be able to provide processes with the required resources.

Process migration is a well-explored problem with multiple existing solutions[2, 3], which can be split into three categories: migration-aware operating systems[4, 5, 6], migration-aware processes[7], and process check-pointing[8, 9, 10, 11]. Of these, I consider only the latter two, as consumers would be unlikely to change operating system just to use the Dropcycle service.

Migration-aware processes are able to transfer themselves from one machine to another, often by sending a copy of themselves to the new machine and then passing their current state to the copy. While this approach makes process migration straightforward, it would require users to be able to rewrite their programs to work with Dropcycle. Dropcycle would also have to provide libraries for process migration in multiple languages and for multiple operating systems.

State check-pointing requires more work during migration, but is almost unnoticeable to the user. The user process is stopped, and a snapshot is taken of the current process state - its register values, virtual memory, any open file descriptors or network sockets, and so on. This information is then transferred to the cloud and the process is restarted there. As existing solutions for process check-pointing are Linux-only and incomplete, Dropcycle would have to implement custom solutions for each operating system that they plan to support. All of the major operating systems (Windows, Mac OS X, and Linux) provide ways to access the necessary state information (through drivers, kernel extensions, and kernel modules[1] respectively.)

Once the process is ready to be transferred, a suitable host must be provided for it. This is a difficult task, as clients could have any combination of

[1] It is also possible to check-point a process in Linux entirely from user-space[15].

operating system and system architecture. Additionally, one of the requirements for Dropcycle is to have all of the cloud nodes running Linux. There are two approaches to solving this problem. Firstly, Dropcycle could choose to only support processes running on Linux on a particular architecture[2], and run the processes directly on their cloud servers. This would be simple in some regards, such as only requiring Dropcycle to create a Linux version of their various supporting programs, but it would limit the potential user-base to users who run Linux.

The other option for hosting Dropcycled processes is to use virtualization to support any combination of operating system and system architecture. Unfortunately due to legal reasons it is not possible to virtualize Mac OS X on non-Apple hardware[17]; since Dropcycle is required to use Linux servers this means that it would not be possible to support Mac OS X. Control of the virtual machines would depend on where the servers were located. If Dropcycle uses its own servers, multiple virtual machines could be started per physical server, and controlled via tools such as Vagrant[18]. On the other hand, if a cloud provider such as Amazon was used, Dropcycle would need to create a tool to programmatically spin up custom instances[19] with the appropriate virtual machines.

There is no point in migrating a process to the Dropcycle cloud unless more of the resource it is lacking can be provided. This means that there must both be a way to determine how much of a resource the process requires, and a way to find a host for the process that can provide this resource. The former problem is already solved as Dropcycle has created a service that can detect when a process has exhausted local resources. This means that it must track them somehow, and so resource requirements could be estimated by extrapolating past usage. Locating a host for the process would be more complicated - while just finding a server (or creating a virtual machine) with enough resources for the process might be straightforward, efficiently balancing the Dropcycled processes so resources are not wasted is a complex constraint satisfaction problem.

The final issue in process hosting is where the servers should be physically located. Here there are two choices: Dropcycle can either choose to maintain their own set of servers, or they can employ the services of an existing cloud computing infrastructure (such as Amazon EC2 or Voxel VOXCloud[14]). This involves a trade off between several key features: the costs, the scala-

[2]Most likely x86, as it is the architecture supported by most Linux distributions[16].

bility, and the possibility of Dropcycle-specific optimizations.

In an ideal situation, running their own server farm would cost Dropcycle up to half the price of an equivalent Amazon EC2 setup[20]. However, should significant hardware problems arise, the cost of self-managed servers would rise, whereas the cost of using EC2 would not. A more important issue is how well each option can scale. A self-managed server farm would be slow to scale, as new machines would have to be purchased and installed to support more users. On the other hand, using a cloud service provider means scalability is fast and easy - you can just keep spinning up new instances to meet new requests. A downside of using a cloud service provider would be how well you could optimise for Dropcycle-specific requirements. For example, it would not be possible to run multiple virtual machines on a single instance if using EC2, and so there would be a less granular control of resource limits available. Self-managed servers would not have this problem - the user would be able to specify exact per-process resource limitations. A self-managed farm could also be optimised for intra-Dropcycle process migration and communication (i.e. by placing all of a users processes in physically similar locations.)

2.2 File-System Issues

The main issues regarding file systems are how to store Dropcycled files in order to allow both cloud and client processes to access them, and how to deal with Dropcycle processes which attempt to access client-local files.

Both processes that have been Dropcycled and ones that are running on the client may want to access Dropcycled files. There are three possible solutions for storing the Dropcycled files: they could be copied from the client to the cloud, and a synchronisation policy put in place to mitigate concurrency issues; they could be kept on the client and accessed from the cloud; or the Dropcycle folder could be part of a distributed file system, which both the client and cloud processes could access. Duplicating the Dropcycled files on both the client and server side is similar to how Dropbox works - it would require a concurrency model to make sure that files were kept synchronized, which would cause a lot of network traffic. Keeping the files solely on the client would remove the need to implement a custom concurrency model, but would require Dropcycle to wrap the operating system file commands, and would be reliant on the client staying up - again requiring a lot of network traffic. The final option is to never store the files locally, instead having the Dropcycle folder be a distributed file system directory. While this would not solve the network traffic problem, it would move it to a set of file servers which are specifically built to handle such traffic. It would also remove the need to devise a custom concurrency model - the synchronisation method would be determined by the choice of file system.

Dropcycled processes may also attempt to access non-Dropcycled files, which would not exist on the cloud server where the process was residing. As this is outside the specifications, it would be acceptable to simply disallow this case. Alternatively, supporting non-Dropcycled file access would require wrapping the operating system file commands as above to do a network access if the file is not found locally. This would also raise the question of what to do in the case of the client going offline, which is covered in more detail below.

2.3 Client-Side Issues

Issues also exist on the client side. Users must be able to control the Dropcycled processes, and set per-process limits dynamically. A decision must also be made on what to do if the client machine goes offline while Dropcycled processes are running.

There are two options for controlling Dropcycled processes from the client. The most intuitive solution (for the user) would be to insert a dummy process in place of the migrated process, which would pass user signals and input to the cloud, and pass output back to the user. However, this would require migration-aware processes, as otherwise the dummy process would not be guaranteed to have the same *pid* as the migrated one. A simpler though less intuitive solution is to provide the user with a custom program to control their Dropcycled processes.

The difficulty of dynamically setting process limits would depend on how the Dropcycled processes were being hosted. If Dropcycle uses virtual machines running on their own servers, then it is possible to set the virtual machine limits so that the processes cannot use too many resources. To dynamically alter these limits, a new virtual machine would have to be created, and the Dropcycled process migrated using the same method as before. This solution would be easy to implement, but changing the limits would be slow. If a cloud provider was being used, it would be more difficult to set fine grained process resource limits, as providers do not offer that level of control. For example, Amazon EC2 has only 11 instance types[21], so the user would only have 11 choices of resource limits. Having multiple processes running per virtual machine, or running processes run straight on self-managed servers would allow for more fine-grained resource control, but a monitoring program would need to be created to stop processes using more than their allowed resources. Dropcycle has already developed a program for monitoring malicious processes, so all that would be needed would be to set the limits for this program dynamically.

If a client goes offline while it has Dropcycled processes, there are three options. Firstly, it could simply be decided that all of the client's Dropcycled processes should be cancelled. This would be straightforward, but would defeat the point of having cloud computation. Alternatively, processes could simply be allowed to continue running when the client fails. This is again straightforward, but raises questions of what to do when the process takes some action that interacts with the client. Finally, Dropcycled processes could be stopped and check-pointed when the client fails, and restarted once the client re-establishes a connection. This would avoid the problems with Dropcycled processes attempting to interact with an offline client. However, it may also be against the point of cloud computation. Detection of client failure is straightforward; either Dropcycle can routinely ping the client, or a Dropcycle program running on the client can routinely ping a server.

2.4 Communication

There are two issues that need to be dealt with regarding communication in Dropcycled processes - communication between processes, and external network access.

The difficulties posed by inter-process communication depend on what sort of communication is being used. There are four main types of inter-process communication: files, sockets, pipes, and shared memory. File-based communication would depend on whether access to client-local files is allowed for Dropcycled processes. If it is then there is no limitation on file-based communication. However, if access to client-local files is restricted, then file-based communication will only continue to work if the file is in the Dropcycle folder. It is also possible to support socket-based communication, by inserting a relay process in place of the Dropcycled process that listens on the same socket(s) as the migrated process and relays received messages to the cloud. This would be a simple extension to the migration code. Pipe-based communication can be supported using tools such as netcat[22] and socat[23], which can connect pipes to network sockets (where they would then be relayed to the cloud), although this would require migration-aware processes, to pass the pipe onto the relay process. The last form of inter-process communication, shared memory communication, is more difficult. Maintaining a consistent shared memory state from a client process to a Dropcycled process, or from one Dropcycled process to another, would require either using a distributed address space or finding some method to synchronize the address spaces of the communicating processes. Either of these methods would remove the most important feature of shared memory communication - speed. As such, an equally viable solution would be to just disallow it.

Processes may also need access to an external network, either to retrieve data, or to provide a service to outside agents. Providing a Dropcycled process with incoming network access is straightforward, whether the processes are being hosted on a physical server or on a virtual box - in either case the host will have access to the internet. Supporting a process that is providing a network service is more difficult. While the implementation would be straightforward (using another relay process to relay requests from the client to the cloud and then to feed responses back), it would slow the service down, and a decision would have to be made about what to do if the client went offline - it might be the case that the server process would just continue to run idly forever, wasting the clients purchased resources.

3 Proposed Solution

As in the previous section, the proposed solution for Dropcycle is divided into four categories.

3.1 Process Moving & Hosting

My advice to Dropcycle for process migration would be to use the state check-pointing method described above. While it is more complex than using migration-aware processes, it means that (almost) any process could run on Dropcycle without alteration, which is important for the target audience. The existing work on Linux would provide a starting base for Dropcycle to develop their own software.

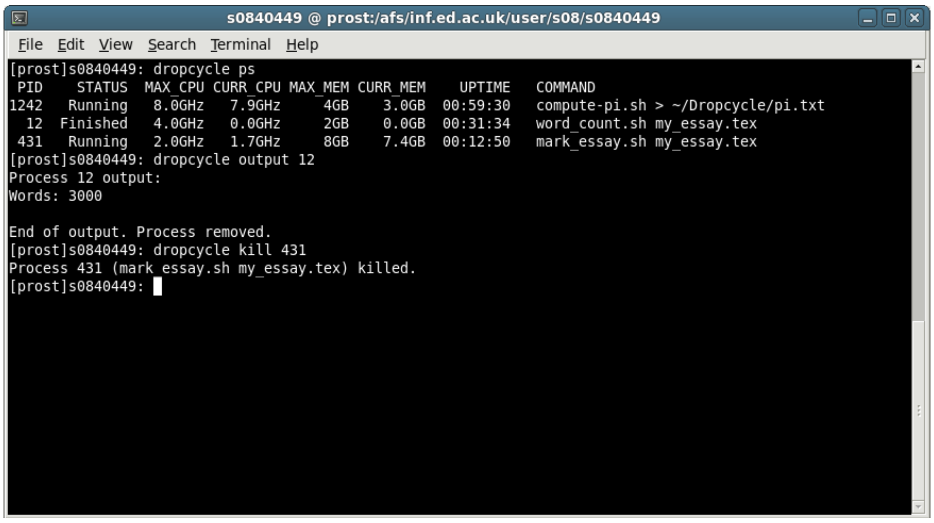
To host the processes on the cloud Dropcycle should use a limited set of virtual machines to provide for the most likely customers. This would include the Linux and Windows operating systems, and the x86 and amd64 architectures, as they cover the majority of user systems. Virtualization allows Dropcycle to support a large number of potential users on a single set of servers, and also brings other benefits which will be explained below. Each virtual machine should support a single process. Although this is less efficient than running multiple processes per virtual machine it provides inbuilt security, preventing cross-process contamination.

As mentioned above, I would suggest that Dropcycle use their existing tool to estimate resource needs, starting with linear extrapolation and then investigating machine learning. A minimum buffer could be defined to make sure that the linear extrapolation does not underestimate the resource requirements.

The choice between using self-managed servers and a cloud service provider is a difficult one, but I would advise Dropcycle to choose to use a provider like Amazon EC2 over managing their own servers. Although the cost would likely be higher, EC2 provides an instantly scalable service, which would be important for a service like Dropcycle who may find that many of their clients need their services at once. The different sizes of instances that Amazon provides should allow the user reasonable control over how many resources to give each process. To control the virtual machines Dropcycle should have a master server (also running on EC2) that is responsible for receiving processes from clients and starting virtual machines for them. It would also be responsible for dealing with the possibility of instances failing. Instances should routinely (say every 10 minutes) send the master server a snapshot of the current process state. If the master does not hear from a virtual machine, it assumes that it has failed and launches a new virtual machine with the last known process state. This should allow for the case when Amazon EC2 instances fail. Should the master itself fail there should be a backup master that can swap in - the original master would automatically reboot and become the backup.

3.2 File-System Issues

Dropcycle should use a distributed file system for the Dropcycle folder, with a directory per user. Both the user's local machine and each virtual machine running a user's process should be able to access that user's directory. Using a distributed file system provides Dropcycle with an inbuilt concurrency model, and removes the need to move files from the client machine to the cloud. The particular choice of distributed file system would depend on the likely file sizes


s0840449 @ prost:/afs/inf.ed.ac.uk/user/s08/s0840449
File Edit View Search Terminal Help

```
[prost]s0840449: dropcycle ps
 PID    STATUS  MAX_CPU CURR_CPU MAX_MEM CURR_MEM    UPTIME   COMMAND
1242   Running   8.0GHz   7.9GHz     4GB    3.0GB  00:59:30   compute-pi.sh > ~/Dropcycle/pi.txt
  12  Finished   4.0GHz   0.0GHz     2GB    0.0GB  00:31:34   word_count.sh my_essay.tex
 431   Running   2.0GHz   1.7GHz     8GB    7.4GB  00:12:50   mark_essay.sh my_essay.tex
[prost]s0840449: dropcycle output 12
Process 12 output:
Words: 3000

End of output. Process removed.
[prost]s0840449: dropcycle kill 431
Process 431 (mark_essay.sh my_essay.tex) killed.
[prost]s0840449: 
```

Figure 1: A mock-up of the potential Dropcycle interface for the client.

that users would be dealing with. Given the target audience I believe that the files sizes would not be large enough to justify the use of HDFS[24], so I suggest that Dropcycle use AFS[25] instead. The downsides of AFS are not a large problem for Dropcycle: the last-write-wins model of AFS's session semantics mimic the behaviour that the user would expect from a 'normal' file system, and the directory-based permissions are exactly what is required for a directory per user. This approach also means that the user should no longer be charged for per-process secondary storage. Instead, they should pay a set monthly fee for globally available secondary storage.

For the initial launch, Dropcycle should disallow access to client-local files for Dropcycled processes. It is an unnecessary complication - the specification clearly says that files should be in the Dropcycle folder.

3.3 Client-Side Issues

To control Dropcycle processes from the client I would advise Dropcycle to develop a custom program. This allows the user to control their programs as is nearly normal, and is the best that can be done for migration-unaware processes. A mock-up of what this service may look like is given in figure 1.

For dynamic process limits, Dropcycle should allow the user to choose from the range of instance limits given by Amazon. This does not allow for fine grained resource control, but there is a reasonable range, including high computation and high memory instances. A monitor program similar to the one on the client machine may be run alongside each Dropcycled process, so that when it nears the limits of its current virtual machine it can be migrated to one with more resources, assuming that it has resources left to use.

If the client machine goes offline, its Dropcycled processes should be allowed to run as normal until they attempt an impossible action, such as inter-process communication or reading from stdin. At that point they should be state check-pointed and frozen until the client reconnects. This allows processes that are pure computation to keep running, while user-centric processes will be paused until the user can return.

3.4 Communication

Dropcycle should allow but not support process communication. This means that file-based communication may work (if the processes use a file in the Dropcycle folder), and socket communication will function (as the migration process should set up socket forwarding for the Dropcycled process), but pipe-based communication and shared memory will not work. This is an acceptable solution as if the user is using multiple communicating processes then they are likely heading towards full distributed computation, and should look at running on a cluster or on a cloud service provider.

Like with process communication, Dropcycle should allow but not support network access. Incoming network access will be fine, as each virtual machine will have access to the internet, but if the Dropcycled process provides a service then it will only function as long as the client is on-line. If the client goes offline then the Dropcycle program on it will be unable to forward requests and responses to and from the process, and so the process will idle forever as long as it does not attempt any illegal actions (as described above). This will cost the user, but this is acceptable given that hosting a process that provides a network-based service is outside of the target functionality for Dropcycle. A web interface should be provided so that the user can cancel their processes without needing access to the original client machine.

4 Conclusion

I do not believe that Dropcycle is a good idea for a start up, for two reasons. Firstly, the technology involved in Dropcycle is very complex and still immature; many of the problems examined here have no feasible solution that would cover all edge cases and provide for every user. Additionally, there are multiple problems that have not been covered here which would also need solutions: security of Dropcycled processes, process groups and hierarchies, optimising the Dropcycling of processes, to name a few. Secondly, distributed computing is simply too cheap and readily available for there to be any middle ground for Dropcycle to target. Computations are either small enough to run on a local machine (which are almost as powerful as any server Dropcycle can provide), or they can be easily and cheaply converted to run on a distributed system such as EC2.

References

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

[2] Smith, J., *A Survey of Process Migration Methods*, May 2001

[3] Milojicic, D. S., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S., *Process Migration*, August 1999

[4] The MOSIX Operating System, `http://www.mosix.org/`

[5] The Sprite Operating System, `http://www.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html`

[6] The X-Series Operating System (XOS), `http://www.crossbeam.com/products/xos/`

[7] Bharat, K. A., and Cardelli, L., *Migratory Applications*, SRC Research Report 138, February 1996

[8] Bricker, A., Litzkow, M., and Livny, M., *Condor Technical Summary*, January 1992

[9] CryoPID, `http://cryopid.berlios.de/`

[10] ckpt, http://pages.cs.wisc.edu/~zandy/ckpt/

[11] Berkeley Lab Checkpoint/Restart (BLCR), https://ftg.lbl.gov/projects/CheckpointRestart/

[12] VirtualBox https://www.virtualbox.org/

[13] VMware http://www.vmware.com/

[14] VoxCLOUD http://www.voxel.net/voxcloud

[15] Distributed MultiThreaded CheckPointing (DMTCP), http://dmtcp.sourceforge.net/

[16] Comparison of Linux Architecture Support, http://en.wikipedia.org/wiki/Comparison_of_Linux_distributions#Architecture_support

[17] Legality of Mac OSX Virtualization, http://www.macrumors.com/2011/07/01/os-x-lion-allows-running-multiple-copies-on-the-same-machine-virtualization

[18] Vagrant, http://vagrantup.com/

[19] EC2 Custom Virtual Machine Importing, http://aws.amazon.com/ec2/vmimport/

[20] A Comparison of Self-Managed servers against EC2, http://blog.rapleaf.com/dev/2008/12/10/rent-or-own-amazon-ec2-vs-colocation-comparison-for-hadoop-clusters/

[21] EC2 Instance Types, http://aws.amazon.com/ec2/instance-types/

[22] Netcat, http://netcat.sourceforge.net/

[23] Socat, http://www.dest-unreach.org/socat/

[24] Hadoop Distributed File System, http://hadoop.apache.org/hdfs/

[25] Andrew File System, http://www.cmu.edu/corporate/news/2007/features/andrew/what_is_andrew.shtml