AWSAC: Amazon Web Services for ATLAS Computing

documentation

Jan-Philip Gehrcke

Max-Planck-Institut für Physik, München
Universität Würzburg

last amended: December 16, 2008

CONTENTS

CHAPTER

ONE

ABSTRACT

An **ATLAS Software Release** is a software collection to perform computing within the scope of the ATLAS experiment at the *Large Hadron Collider* (CERN, Geneva). This documentation describes how to perform computing with an *ATLAS Software Release* in Amazon's **Elastic Computing Cloud EC2**. Therefore, an **Amazon Machine Image** (AMI) on the basis of the standard ATLAS platform **Scientific Linux 4** (SL4) was created. The development is shown in detail. A proof-of-principle **job system** was developed. Jobs can use the software in the normal way. The output of a job is exported to Amazon's **Simple Storage Service S3**, before the corresponding virtual machine in the cloud is terminated. Job status information is transferred to Amazon's **SimpleDB** service. Job sessions are controlled and monitored by a client using **Python** scripts implementing the **Amazon Web Services** API via the **boto** library working together with scripts embedded in the SL4 AMI. The experience with setting up and operating the described system using standard ATLAS job transforms is reported.

It is shown, that **ATLAS Cloud Computing** is a promising substitute for classical *ATLAS Computing*. The basic services a computing cloud should deliver for *ATLAS Cloud Computing* were elaborated. The benefit of a **general Cloud Computing API** with different possible clouds at the back end is discussed. It seems possible to set up an *EC2* style cloud in the future, so that the advantages of *Cloud Computing* would be available for the price of own hardware.

CHAPTER

TWO

INTRODUCTION

Cloud Computing with virtualization could make ATLAS Computing much more reliable and robust. *Cloud computing* centralises the hardware and decentralises software. It hides complexity from the user of the services. Services are made available in a very flexible way via virtual servers. This results in less administration effort and staff costs in computer centres. So the LHC physicists should find ways to use *Cloud Computing* to their advantage. Using Amazon Web Services (AWS) is one obvious possibility to determine the possibilities for *ATLAS Computing* in a cloud.

In summer 2008 *Prof. Dr. Thomas Trefzger* (Uni Würzburg, Fakultät für Physik und Astronomie) sent me (Jan-Philip Gehrcke) to the Max-Planck-Institut für Physik in Munich, where I stayed for six weeks following *Stefan Kluth*'s consideration to use AWS as a platform for *ATLAS Computing*. The idea was to especially use Amazon's so-called «Elastic Computing Cloud» (EC2) for simulation, reconstruction and other applications an ATLAS Software Release delivers. We call the whole topic **AWSAC**: *Amazon Web Services for ATLAS Computing*.

At first we configured a Linux system that fulfils some special conditions simultaneously: the complete ATLAS Software Release and the Amazon EC2 AMI Tools have to run on it properly and it has to run on the virtual machines of the *Elastic Computing Cloud* correctly itself, using the «Xenified kernel» Amazon delivers. The way we did it is described in this documentation in every detail.

Then we started developing **AWSACtools**, providing an *ATLAS Computing* job system. This system takes advantage of the possibility to start up as many virtual machines in Amazon's *Elastic Computing Cloud* as you need. *AWSACtools* consist of a client- and a server component to manage *ATLAS Computing* jobs. One job is basically one shell script completely written on your own (you can use the *ATLAS Software Release*'s commands in there, of course). It is executed as root on a virtual machine in the cloud as a sub process of *AWSACtools*' server component. To control jobs' statuses and to receive jobs' results with the client component, *AWSACtools* use other *Amazon Web Services* like Amazon SimpleDB and Amazon Simple Storage. *AWSACtools* are documented here in every detail, too.

The proof-of-principle job system shows, that the approach of *Cloud Computing* for *ATLAS* can make things really easy. At the end of the documentation the basic services a computing cloud should deliver at least for an *ATLAS Computing* job system are discussed. It is argued,

why a general *Cloud Computing API* with different possible clouds at the back end is highly desired.

CHAPTER

THREE

WHAT ARE AMAZON WEB SERVICES?

Amazon Web Services (AWS) accrued in early 2006. It is part of the well-known electronic commerce company Amazon.com. AWS emerged from Amazon.com's enormous global computing infrastructure, that has been built up for over 13 years. The objective is to **offer flexible IT infrastructure services from one source (the cloud)** for customers like companies of all sizes. These services include e.g. compute power, storage and data bases. The most important two principles of all *Amazon Web Services* are:

- **You pay only for what you use.**
- **You can instantly get as much from everything as you need.**

Let me quote Amazon: «Using *Amazon Web Services*, an e-commerce web site can weather unforeseen demand with ease; a pharmaceutical company can "rent" computing power to execute large-scale simulations; a media company can serve unlimited videos, music, and more; and an enterprise can deploy bandwidth-consuming services and training to its mobile workforce.» https://aws.amazon.com/ is the home of *Amazon Web Services* (AWS), whereas "home" has two different meanings:

- By using this URL in a browser, you can get information about AWS, manage your AWS account, browse documentations, examples, tutorials, articles and visit forums.
- By using this URL (and sub domains) in any high-level programming language, you can control any of the *Amazon Web Services*. The whole steering communication is HTTP based and directed to the named web server at aws.amazon.com.

The most important parts of AWS are currently placed in three big computer centres at the east cost of the USA. These centres are at three physically totally different places and form three so-called availability zones.

In the following parts, I will name and shortly describe the three services that are used by *AWSAC*.

3.1 Simple Storage (S3)

S3 is Amazon's storage service. Its most important features are:

- Simple HTTP based API to store and retrieve any amount of data, at any time, from anywhere on the web.
- Very high bandwidth and fast access from within the cloud.
- Highly reliable (automatically mirrored between the three availability zones).

S3 **is not a file system**. It has a **bucket / object system**:

- One AWS account can own up to 100 so-called *buckets*. The namespace of these *buckets* is a public space, so one *bucket* name only exists one time for all AWS accounts.
- A "file" at *S3* is a so-called *object*, placed in a *bucket*. The "filename" of the *object* is the so-called *key*. Objects may contain from 1 byte to 5 gigabytes of data. The number of *objects* you can store in a *bucket* is unlimited.
- *S3* has an entire access control system for *objects*. Rights can be granted to specific users or to the public.

The pricing is by storage used, traffic between *S3* and "the internet" and requests. Traffic in the cloud (e.g. between *S3* and *EC2*) is for free.

More detailed information about *S3*: http://aws.amazon.com/s3/

3.2 Elastic Computing Cloud (EC2)

EC2 is Amazon's computing power service. It provides flexible (resizeable) computing capacity in the cloud. *EC2* is a **virtual computing environment**. It allows you to launch and terminate virtual machines of **individually configurable operating systems** (standard: Linux systems; possible: Windows systems). Such a custom system can be stored in a so-called *Amazon Machine Image* (AMI) containing e.g. custom applications, libraries, data and associated configuration settings. AMIs are stored encrypted on *S3*. **Virtual machines at EC2 are called instances**. Launching instances of a custom AMI makes it possible to get virtual servers with a custom application environment.

Important features of *EC2* are:

- Flexibility / Elasticity: *EC2* can **launch or terminate as many instances (virtual servers) as you want at any time** within minutes.
- *EC2* is controlled via easy HTTP based APIs. So an application can automatically scale itself up and down depending on its needs.

- *EC2* grants **root access to the virtual machines** and access to the console output.
- One can choose the virtual hardware before launching an instance.
- One can use so-called public AMIs of pre-configured systems or create own private AMIs.
- After creating an own AMI one can make it public so that others can launch their own instances of it.
- So-called *Elastic Block Stores* (EBS) provide persistent storage for *EC2 instances*. **EBS volumes exist independently from the life of an instance.** This is convenient, because the so-called *instance storage* - that comes automatically with an instance - gets lost when the corresponding instance shuts down. **EBS Volumes provide a real file system and a higher performance than S3**.

Pricing for *EC2* is basically per "instance-hour" - whereas the *instance type* (the virtual hardware) plays an important role - and traffic between *instances* and the internet. The costs for *EBS* and other "subservices" (like *Elastic IP*) are calculated additionally.

More detailed information about *EC2*: http://aws.amazon.com/ec2/

3.3 Simple DB

SimpleDB is Amazon's database service. As you can guess from the name, it is constructed to be simple. The service is intended to be used from within the cloud, i.e. from *EC2* instances.

Important features:

- Provides the core functionality of a database: real-time lookup and simple querying of structured data **without the complexity of a standard relational database**.
- *SimpleDB* automatically indicates the data accordingly.
- *SimpleDB* is fast (from within the cloud).
- The data is stored redundantly across multiple servers and data centres (like data on *S3*).

More detailed information about *SimpleDB*: http://aws.amazon.com/simpledb/

CHAPTER

FOUR

WHY AMAZON WEB SERVICES FOR ATLAS COMPUTING?

To answer this question, I at first have to argue, why scientific computing like *ATLAS Computing* should profit from *Cloud Computing* with virtual machines in contrast to e.g. the *LHC Computing Grid* (LCG) approach.

4.1 Why Cloud Computing with virtual machines?

In computer centres of the LCG the need for software maintenance (operating systems and job systems) is very big, because unstable software and particularly user mistakes often lead to crashed systems. Because of this, **the personnel effort and costs for administration and maintenance in the LCG is big**.

So, what are the advantages of working with **guest operating systems** on **virtual machines** (VMs)?

- The user does not touch the **host operating system**.
- Different VMs on the same hardware work independently.
- The user can work as root without worry.

In other words:

- The user is not able to damage the host OS (bad intentions excluded).
- If one VM crashes, the other VMs on the same hardware proceed working.
- The user may "destroy" one VM, but he can just relaunch it (takes advantage of the **machine image concept**)

So, when the virtual machine monitor (the software layer providing the virtualization) runs stable, there ideally should be nothing, that could crash the host OS. From the computer

centre staff's point of view, the software (the host OS) then nearly is **maintenance free**. **With virtual machines in a cloud, computer centres would almost only have to care about the hardware, which would reduce personnel effort and expenses!**

You perhaps ask yourself, if there is a loss of performance, when hardware stressing jobs run on virtual machines. The solution is the open source paravirtualization Xen:

> «Through paravirtualization, Xen can achieve high performance [...] which is notoriously uncooperative with traditional virtualization techniques.»

This requires the guest operating system to be modified in a special way, which is not a big problem, as we can see using the example of *Amazon Web Services*. AWS uses *Xen* virtualization for *EC2* and because of this, there the guest operating systems need a so-called «Xenified kernel».

Hence, there are various advantages, *Cloud Computing* with virtual machines brings along.

4.2 Why Amazon Web Services?

The first reason is that a reliable technical infrastructure to give *Cloud Computing* a try is currently only commercially available. The leading company in this sector is *Amazon* with its *Amazon Web Services*. Additionally, Amazon really is interested in working together with science. They sent out representatives / consultants to the Max-Planck-Institut, to consider our needs and to work on an individual solution for special applications in science. So, for *Cloud Computing* research purposes, AWS is ideal.

When calculating and comparing the costs for computing with AWS and with own hardware, it is obvious, that AWS is more expensive. **It is definitely excluded and not the intention to e.g. move ATLAS Computing completely to AWS.**

But there is another serious advantage, *Amazon Web Services for ATLAS Computing* brings along: imagine, there is a deadline and you have to get e.g. simulation results until a fixed point in time. Additionally imagine, that your classical computing possibilities run out of capacity and it is foreseeable, that computing will not finish. Then **it might be of very big value for you, to occupy a huge amount of instances at EC2, so that you can meet the deadline**.

The described case is exactly the case AWS is designed for. It offers flexible, scalable and elastic computing infrastructure, so that a company or even science gains planning reliability. Remember, that the AWS user only pays, what he uses!

In summary, **AWS offers a very good base to develop Cloud Computing for scientific applications**. AWS is promising in respect of missing computing capacity. Those peaks of desired computing power can be satisfied using AWS for a short time. The resulting cost may be less important than meeting a deadline.

4.3 A profitable mixture - AWSAC's main motivation

Let me summarize:

- *Cloud Computing* using virtual machines would reduce personnel effort and cost.
- Computing with AWS is more expensive than with own hardware.
- AWS offers a great possibility to satisfy peaks of desired computing power

These statements implicate, that **the optimal solution would be an own computing cloud for the price of own hardware in combination with a standard Cloud Computing API** that is able to control the own cloud and additionally e.g. AWS. Then the user is able to decide, in which cloud his jobs should run. **Switching between clouds becomes very easy, resulting in a very convenient solution to balance out peaks of desired computing power**.

This would be a very profitable mixture and research on this topic is exactly **the main motivation of the AWSAC project**.

CHAPTER

FIVE

UNDERSTAND AND CONTROL AWS

If you want to get in AWS and want to understand the things that are important for you, then there are many helpful sources. At first you have to know which special services you need for your application. A summary of all *Amazon Web Services* can be found here. For *AWSAC* the most important services are *EC2* and *S3*. *SimpleDB* may perhaps be called "bonus". *EC2* is much more complex than *S3*. So for me, understanding *EC2* was of top priority at the beginning. Hence, I worked myself through Amazon's EC2 Resources. There you can find the official documentation (including the very essential Developer Guide), great articles, helpful tutorials and useful examples. Additionally, the AWS Forums are very convenient for special problems and questions (great support!).

To spare you to read through everything, in this chapter I will explain the technical facts that you should have in background while reading this documentation.

5.1 Manage all services via HTTP

AWS is using a so-called Web Service Description Language (WSDL) that strictly defines operations and how to control their services. As stated before in *chapter 2*, every service is controlled via HTTP. More exactly, a special language meeting the definitions in the WSDL must be spoken over HTTP. For such a special language a XML structure is convenient. This language is spoken in a conversation between the client (the one that has a special **request**) and the server aws.amazon.com (the one that elaborates a corresponding **response**). HTTP is used as a protocol to arrange this conversation, since **HTTP natively is a request/response standard between a client and a server** and it should be accessible from any internet connection.

Consider a client that wants to perform an action on AWS, e.g. on *EC2*. An action encapsulates the possible interaction between the client and *EC2*, consisting of a request and response **message pair**. Then, simply expressed, the *Web Services Description Language* (WSDL) strictly defines the special structure of this message pair. A special request from a client must meet the corresponding entry in the WSDL. Then either AWS follows this request and performs the requested action; or the request will be declined with a specific error code in the response. The response, in case of success or failure, is strictly defined by

the WSDL, too.

The set of possible actions or valid requests defined by the WSDL for e.g. *EC2* builds the **EC2 Application Programming Interface** (API). One requested action is called **API call**.

The AWS API is subject to constant enhancement. Sometimes a new API version is released. Normally all changes are downwardly compatible; but to avoid any problems, each API version has its own documentation and, most important, the client is able to define the API version its request should be examined with.

5.1.1 Example message pairs

In this part I want to be more precise on the concrete language spoken in a conversation between the client and the server (the language that is transported by HTTP). The request may be formed in two different languages, while the response is always the same language.

Now let me illustrate the two possible the request/response systems, that differ in the form of the requests: Imagine you would like to create a new *Elastic Block Store* volume of 800 GiB in the availability zone "us-east-1a". Then the two possible requests - without authentication and other security overhead - look like

```
<CreateVolume xmlns="http://ec2.amazonaws.com/doc/2008-08-08">
  <size>800</size>
  <zone>us-east-1a</zone>
</CreateVolume>
```

in XML form (the so-called SOAP API) or

```
https://ec2.amazonaws.com/
?Action=CreateVolume
&Size=800
&Zone=us-east-1a
```

in HTTP Query-based form (using standard GET or POST methods to submit parameters)

For both of these two requests the response - in case of success - is **in XML form** and looks like

```
<CreateVolumeResponse xmlns="http://ec2.amazonaws.com/doc/2008-08-08">
  <volumeId>vol-4d826724</volumeId>
  <size>800</size>
  <status>creating</status>
  <createTime>2008-05-07T11:51:50.000Z</createTime>
```

```
    <zone>us-east-1a</zone>
    <snapshotId></snapshotId>
  </CreateVolumeResponse>
```

As you can see, AWS informs you in detail about the requested process. The response always contains all information that you perhaps could need to work on. In this example case, the most essential information is the unique volume ID of the new EBS volume, which you need to e.g. attach the volume to an *EC2* instance.

Example requests and corresponding responses of all *EC2* API calls are listed in the EC2 Developer Guide - API Reference (for both SOAP and Query API).

5.1.2 Authentification and security

The communication with AWS takes place between a managing client and https://aws.amazon.com. Hence, the communication itself is encrypted.

To ensure that only you can perform actions with your account, every request contains authentication parameters. In the case of the Query API, the two important elements of these auth parameters are

- the so-called **Access Key ID** (public identifier of your account)
- a hash that was built locally from the request data itself in combination with the so-called **AWS Secret Key**.

Then AWS reads out the Access Key ID from the request, looks up the corresponding Secret Key (from a database) and recalculates the hash with the same algorithm as the client did. When the two hashes (the one sent by the client and the one AWS calculated) match, the request is identified as valid.

The authentication mechanism is different for different AWS and may differ between the API types, too. E.g. when using the *SOAP API* for *EC2*, security is guaranteed by using an X.509 certificate in combination with an RSA public/private key pair and when using the SOAP API for *S3*, the mechanism is almost the same as described above (for *EC2 Query API*).

To ensure security within the cloud (e.g. between virtual machines of different *EC2* users) and between the cloud and the internet, AWS has an entire configurable Network Security concept.

5.1.3 Using the API for own applications

Since every modern high-level programming language has web service libraries (e.g. for HTTP and XML), it is no problem to implement the AWS API. As a result, there meanwhile

exist different modules/libraries to control AWS for e.g. *Ruby*, *Java*, *PHP*, *Python* and so on. Some of these libraries are official libraries, released by AWS itself. **With the help of these libraries, it is possible to develop big and powerful applications on the base of AWS!**

In the case of *Python*, the corresponding AWS module is third-party (initiated by *Mitch Garnaat*) and called **boto**. Information can be found here: http://code.google.com/p/boto/

I decided to use *Python* with *boto* to control AWS and to build the applications that are now called *AWSACtools*. This is because the *Python* language in my eyes is perfect for doing such scripting things in the easiest way.

5.2 Understand and control S3

S3 is not as complex as e.g. *EC2* and you already know many important things from *the S3 introduction*. But it is convenient to learn some details more about *S3*. At this point I can't summarize it better than Amazon did. Please read the Core Concepts part of Amazon's *S3 Developer Guide*. After this you know everything needed about **the data model** consisting of **buckets**, **objects** and **keys**. Go on reading about Access Control Lists to learn about the mechanism behind **sharing data between AWS users or even with the public**. The rest of the *S3 Developer Guide* also offers interesting stuff.

5.2.1 Manage and monitor *S3* with S3Fox

The S3Fox Organizer is a Firefox plug-in that has implemented the *S3* API. Thus, *S3Fox* has the ability to offer a graphical user interface (GUI) to the *S3* interface. Sometimes it is convenient to visualize the stored data of one or more AWS account(s) at *S3*. Then *S3Fox* offers easy possibilities to check things.

I think that such a tool is really essential, because it may become necessary to e.g. have a look, if a script has really done, what it should have done. So it is great for **monitoring**. But you have to know that *S3Fox* maps the bucket and key structure on a classical directory structure: buckets are the highest directory layer and keys containing "/" are split and treated as directories, too. This sometimes may become confusing.

Moreover, *S3Fox* is helpful to administrate the *Access Control Lists*.

5.3 Understand and control EC2

In this part I will illuminate the relation between the different elements *EC2* consists of and explain how to deal with them using different tools.

5.3.1 Components and nomenclature

Amazon Machine Image (AMI):

An AMI is an encrypted image of almost all files of an operating system, including any user-given data. The encrypted image is divided in *part files*. For each part a checksum is built and logged into a *manifest file*. The *part files* together with the *manifest* file build an *Amazon Machine Image*, which has to be uploaded to *S3* and *registered* (validated), before instances can be launched from this AMI. Building an AMI is done by using special tools, that are described later. After successful registration of an AMI, it gets an individual and unique **AMI ID**. Read what Amazon says about AMIs.

General components:

In the *the EC2 introduction*, I already introduced the most important components of *EC2*. Let me refer to the corresponding part in the *EC2 Developer Guide*, the Components of EC2. Hence, an **Instance** is a running virtual machine in the *Elastic Computing Cloud* that was started from an AMI and **Instance Store** is virtual hard disk space closely connected to an instance which means that it is lost, when the instance terminates.

Let me introduce other important components of *EC2* by means of usage examples. I think that this is the easiest and clearest way. Consider the case you want to start instance(s) with one API call (one so-called **reservation**). Before sending this API call, you have to know exactly what you want:

- **AMI**

  Choose the special system you would like to start up, defined by its AMI ID. Find out the **AMI ID** of your wanted AMI. Only **registered** AMIs can be launched.

- **instance type**

  Choose between various instance types. The type must be adjusted to your needs. These instance types have names like *m1.small* and *c1.xlarge*. A full list of these names can be found in the EC2 Developer Guide - Instance Types. It is important to consider the different cost / performance ratios of the different instance types. If you e.g. like to start up a 32 bit system for high CPU stressing applications, then you should start up *c1.medium* for 0.20 $ per hour instead of *m1.small* for 0.10 $. This is because *c1.medium* has 2 virtual cores with 2.5 *EC2 Compute Units* (ECU) each («One *EC2 Compute Unit* provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor») and *m1.small* only has 1 core with 1 ECU. This is the fifth part of computing power for half of the price in comparison with *c1.medium*.

- **number of instances**

  Carefully choose the number of instances to start up at the same time (with the same API call). AWS has a min/max system: «If [...] *EC2* cannot launch the minimum number [of instances] you request, no instances launch. If there is insufficient capacity to launch the maximum number [...], *EC2* launches as many as possible [...].» It is sufficient to only define the minimal number. This always satisfied my needs.

- **security group**

  As you can read in the Network Security part of the *EC2 Developer Guide*, you can define so-called *security groups*. Each group gets its own firewall settings (everything blocked by default!). If you like to request a reservation (start up instances), you have to specify to which security group the reservation should belong.

- **availability zone**

  The (currently) three availability zones describe different and physically wide divided AWS computer centres. Sometimes it is important to strictly define in which computer centre the instances should start up. The interaction between instances is most efficient, when they are at the same place. And, for e.g. attaching an *Elastic Block Store* volume to an instance, it is even necessary that they are in the same availability zone.

- **keypair**

  If you like to start up a public AMI and wish to log in via ssh, then you don't know root's password. If you like to offer one of your AMIs to the public, then you don't want to distribute root's password. This is inconvenient. The solution are **EC2 keypairs**. *EC2* offers the possibility to create RSA keypairs for you. **Each of EC2 keypairs has a specific name**. *EC2* then keeps the public keys and you keep the private keys (as files) of your *EC2 keypairs*. Delivering one keypair's name within the API call to start instances, you instruct *EC2* to inject the corresponding public key into the instance at boot time. Then you can log in as root using the corresponding private key.

- **userdata**

  The so-called *userdata* can be submitted in form of a *base64* encoded string to all instances that are started by a *RunInstances API call*. All *EC2* instances within this created *EC2* reservation are able to receive this string from an internal *EC2* server with the **static internal IP** 169.254.169.254. In case of the *HTTP Query API*, the userdata is URL limited to 8 kB, so we can't use it to submit files (note: when using the *SOAP API*, *EC2* limits the userdata

string to 16 kB). But we can use it to make essential and necessary information accessible for the instances within the reservation.

Some of the named parameters are optional: you must not define a keypair or userdata; no availability zone leads to a random one, no security group leads to the default group.

> **Note:** Let me again explain the term **reservation**. Every *RunInstances API call* instructs *EC2* to start up one or more instances. All instances started up within this **one** API call belong to the same *reservation*. This *reservation* has a unique **reservation ID**. So all instances just started have the same *reservation ID*. But every instance within a reservation has an own globally unique *instance ID*. Additionally each instance within a reservation has an own so-called **launch index**, starting from `0`. This is important for developing applications, because **using launch-index and/or instance ID is the only way to distinguish between various instances within one reservation**.

Consider the RunInstances API call as valid and the requested instances as started up. If you like to connect to one of your instances, the most important parameter is the **public DNS name**. You e.g. can use this address to connect to your instance via ssh, preconditioned that the corresponding security group the instance is in allows external accesses on port 22.

Elastic Block Store:

As stated above, the **instance storage** is lost, when the corresponding instance shuts down. In the list of instance types you can see, that this **instance storage** is really big. But his doesn't help, if you like to have persistent and fast storage that exists independently from instances. *S3* is not a file system and is sorted out (note: there are third-party possibilities to wrap *S3* and create a pseudo file system on top of it). The solution is **Elastic Block Store (EBS)**. An EBS volume itself can be considered as independent *EC2* component. It can be created at any time (you have to define its size and its availability zone) and deleted at any time. In the meanwhile it may be **attached** to instances (only **one instance at the same time**). Within such an instance an EBS volume appears as hard disk drive that can be mounted and formatted with any file system.

Amazon developed an API call, that invokes a backup of a specific EBS volume to *S3*. Such a backup is called **snapshot**. It is possible to create a new EBS volume from an existing *snapshot* within seconds. Each *snapshot* has a unique ID, the *snapshot ID*.

Read what Amazon says about EBS.

5.3.2 Manage and monitor *EC2*

AWS has implemented its *EC2* API itself using *Java* and *Javascript*. The results are the command line tools **EC2 API Tools** and the Firefox plug-in **Elasticfox**. To create own AMIs, Amazon delivers the **EC2 AMI Tools**, based on *Ruby*.

- **EC2 API Tools:**

  The *EC2 API Tools* provide little programs for the command line for invoking all possible *EC2* API calls. Since they are command line tools, they are useful for scripting. Each little tool with its possible command line parameters is documented in the EC2 Developer Guide - API Tools.

  The *EC2 API Tools* implement the *SOAP API*. As stated above, in *4.1.2 Authentification and security*, this means that the authentication of requests is warranted using a **private key / certificate system**. If you are the owner of an AWS account, **the corresponding files (certificate and key file) can be downloaded from the AWS account management interface**.

  You can get the *EC2 API Tools* here: EC2 Developer Resources - Amazon EC2 API Tools.

- **Elasticfox:**

  Elasticfox grants a graphical user interface (GUI) to the *EC2* API. During development, **Elasticfox became one of the most important tools for me**. Since it knows and controls the complete set of API calls, it replaced the *API Tools* for me in terms of monitoring and manual *EC2* management. It is very helpful to visualize what scripts are doing, since it e.g. displays the state of instances and EBS volumes. Launching and terminating instances becomes very easy, like registering AMIs, creating keypairs, creating and attaching EBS volumes and so on. Try it!

  Since it implements the *HTTP Query API*, it only needs your *AWS Secret Key* and the *Access Key ID* to handle your account.

  Check it out at *Sourceforge*: http://sourceforge.net/projects/elasticfox/ or in the EC2 Developer Resources.

- **EC2 AMI Tools:**

  If you like to create an own *Amazon Machine Image*, you should adopt the help of *EC2 AMI Tools*. They are command line tools, too. Usage is explained in the EC2 Developer Guide - AMI Tools.

  The tools are able to bundle an operating system into an AMI. They provide two ways to do this:

  1. bundle the needed files of a running system into an AMI
  2. bundle a loopback file containing a system into an AMI

After having the image itself, the *EC2 AMI Tools* use your *EC2* private key / certificate to perform the encryption. Then, the encrypted image is divided in *part files*. For each part a checksum is built and logged into the *manifest file*. Additionally, the tools provide the possibility to upload everything to *S3*. *Registering* then is done via *Elasticfox* or the *EC2 API Tools*.

Get the tools from EC2 Developer Resources - Amazon EC2 AMI Tools

CHAPTER

SIX

THE JOB SYSTEM: HOW AWSACTOOLS WORK

In this chapter the **AWSACtools**, comprising an ATLAS job system developed especially for *AWS*, are described in detail. At first, the requirements are derived. Then, the developed job system is described in two detail levels. If you do not understand everything at the first time, then read on. The parts at the end of this chapter (containing voluminous examples) should make things clearer.

Note: There is an extra web page to view and download *AWSACtools*' sources.

Look here: http://gehrcke.de/awsac/permstuff/AWSACtools/

6.1 Job system requirements

In this part, the requirements of the job system are discussed. What possibilities must the job system provide? From the **user**'s point of view:

> The **user** is the one starting and controlling jobs with a **client** software. He must be able to prepare computing jobs of any style with as few as possible restrictions. Therefore he must be able to control the operating system, send any necessary information (input) and to define the results he likes to receive in the end (output). While a job is active, it should be possible to monitor its status.
>
> It has to be possible to start more than one job at the same time: The job system and the client software must provide an easy way to start a whole bunch of jobs - a **job session**.

So our job system must consist of different necessary components that are introduced here.

- **Job:**

  In order to meet the requirements stated above, our **definition of a job** is the following:

One job is a **shell script, executed as root, that runs as its own process** on an instance. The script is **provided by the user itself**. Within this script, the commands of an *ATLAS Software Release* and any user-given file (input) can be used. Any result files can be defined (output).
The job shell script must be controlled by another process in order to e.g. detect errors, which is important to realize monitoring.

- **Input:**

  The job system must be able to deliver various user-given input files that must be accessible from within the job shell script.

- **Output:**

  Output files generated by the job shell script must be received by the job system and saved reliable, e.g. on *S3*. These results must be accessible by the user (the client) in the end.

- **Status information:**

  The job should inform the job system about its status in real-time.

- **Session:**

  A job session consists of one or more jobs, started by the user at the same time. To realize this, we can use the possibility of *EC2 reservations* (start more than one *EC2* instance at the same time). A job session must be identifiable by a unique ID, the **session ID**.

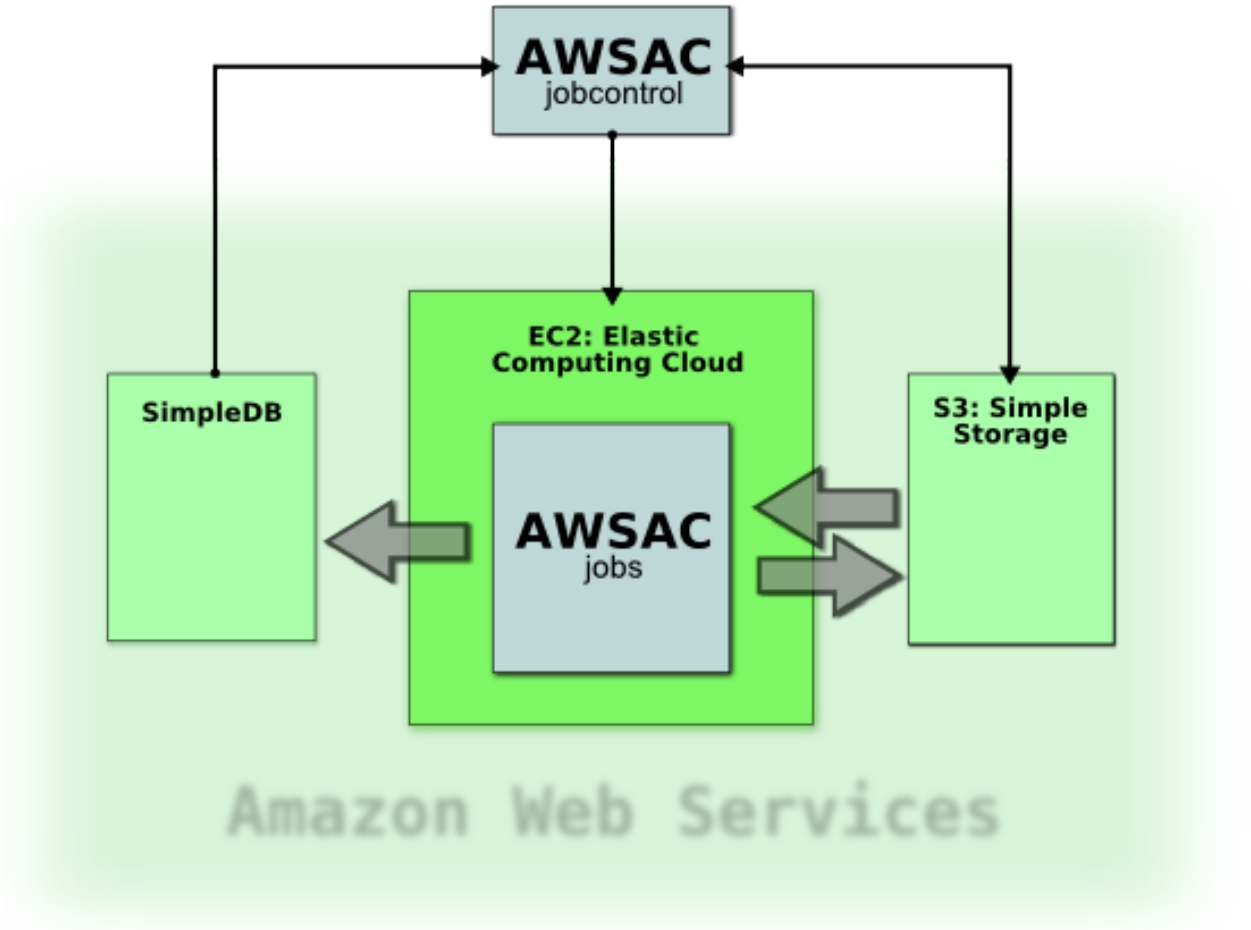
Following these requirements, **AWSACtools**, providing a job system for *ATLAS Computing* with *Amazon Web Services*, were developed.

6.2 The AWSACtools job system

This part describes the developed job system in every detail. The sources of the *AWSACtools* can be viewed and downloaded here: http://gehrcke.de/awsac/permstuff/AWSACtools/ .

6.2.1 Basics

The basic elements of the job system developed are visualized here:



As you can see, three of *Amazon Web Services* are used: *S3*, *EC2* and *SimpleDB*.

The job controlling part of the system is the *client* (*AWSAC jobcontrol* in the picture). It is commanded by the *user*. It has the following tasks:

- start job sessions
- deliver input data for jobs
- monitor jobs' statuses
- receive results and clean up

The *input* is uploaded to *S3*, where it is accessible from *EC2* instances. *S3* is additionally used for delivering an *ATLAS Software Release EBS* snapshot and to save the *output*, which can be downloaded from the client later on. Jobs are executed on *EC2* instances. They send their status information to *SimpleDB*. This monitoring information can be fetched instantly using the client machine.

The system is now described again, but one detail level higher.

6.2.2 Detailed description

The job system delivered by *AWSACtools* consists of various scripts. The main components are shorty introduced here:

- `awsac-session` (*Python*): The client with the tasks described above. Can be started in different modi: `--start`, `--check`, `--getresults` and `--cleanup`.
- `awsac-autorun` (shell script): Hard coded into *AWSAC AMI*; integrated in system start-up - tries to receive session information and initiates long chain of commands.
- `awsac-all-instances-autorun` (shell script): Delivered within job session input. Can be modified by the user - to make the system very flexible. If it is found by `awsac-autorun`, it is executed.
- `awsac-processjobs` (*Python*): Delivered within job session input (can be modified by the user, too). Executed by `awsac-all-instances-autorun`. Executes and controls the job shell scripts; manages *ATLAS Software Releases* (using *EBS snapshots* and *volumes*), monitoring information (using *SimpleDB*) and job output (using *S3*).

In order to make the system more comprehensible, the next paragraphs will describe the procession of an example job session from the beginning to the end in more detail.

1. **The start process on the client:**

   Assume the user prepared all necessary job session information. Together this will be the **session start info file** and the so-called **session archive**. The former defines things like a short session description, the desired AMI (by means of its ID) and the favoured *instance type*. The latter contains the **job definitions file** (described later), the job shell scripts and everything else needed on the instances.

   Assume that this session information is given to the client `awsac-session -start`. When it then starts the desired session, the client at first **generates a new session ID**. Then it **stores the session archive on S3** - using the *session ID* for *S3* key generation.

   The *session ID* and some other selected **essential information is put together in a special userdata string** (see *4.3.1: userdata string*). After accomplishing some checks, the client requests a new *EC2 reservation* with the favoured **AWSAC AMI** and *instance type*, the right number of **instances starting up** and the constructed userdata string. The right number guarantees that each job shell script of the session just started will get its own virtual core and the userdata string ensures that each instance knows which session it belongs to.

1. **Initialization of the session on the instances:**

Instances within this job session are starting up the desired special *AWSAC AMI*. This AMI contains a modified `rc.local`, which is executed after the linux system has booted up and initialized. From this `rc.local` the hard-coded shell script `awsac-autorun` is executed. This happens on **each** instance of the *AWSAC AMI*: it does not make any difference, if the instance is within a job session or not. `awsac-autorun` **tries to receive a userdata string**. If it could be received, it is stored in the so-called **session information file** and the chain of conditional commands goes on with the next steps: parse the string and - using information from the string - **receive the session archive** from *S3* using another small *Python* script. If one of many conditions is not satisfied, the chain stops. `awsac-autorun` is the last hard coded element in the chain. Now we leave the domain of pre-defined things: within a successful session, the *session archive* contains another shell script: `awsac-all-instances-autorun` - if it is found, it is executed by `awsac-autorun`.

As you can see, the job system itself is very flexible: it still does not exist! This means that **the whole job system** - beginning with `awsac-all-instances-autorun` - **can be defined completely by the user**. But don't worry, the job system introduced before was realized (as "an example job system" that can be modified by you using the same AMI).

2. **What do I have to do?**

   Each instance within the session has downloaded and extracted the *session archive*. `awsac-all-instances-autorun` was found and executed. This little shell script is used to initialize the *Python* program `awsac-processjobs`, the real worker of the server-sided part of the job system. The information `awsac-processjobs` must be initialized with is summarized and explained now.

   Each job (defined in the **job definitions file**) should be processed only once. So an instance has to find ans answer on the question «What do I have to do?». It needs a way to distinguish itself from other instances and has to select one or more jobs from the **jobs definitions file** that is not selected by any other instance in the same session. As explained in *the description of EC2 reservations*, each instance within a reservation has its own **launch index**, which is between `0` and `N-1`; in the case of `N` launched instances. Hence, the special **job(s) selected by one special instance within the session depend on the lauch index and on the number of virtual cores of the instance** (each job should get one core for itself).

   But how to find out the *launch index* and the number of cores? The number of *cores per instance* was set by the client (from the *instance type*). It was stored in the *user data string*, which was received by `awsac-autorun`. It is now stored on the instance in the **session information file**. Each instance has its own unique *instance ID*. This one can be fetched from the internal *EC2* server at 169.254.169.254 (like *receiving userdata string*).

`awsac-all-instances-autorun` receives this ID and stores it into the **instance ID file**. Using the *instance ID*, every additional information about an instance (like the **launch index** and e.g. *availability zone*) can be fetched easily within a *Python* script, using *boto*.

Hence, after receiving the *instance ID*, `awsac-all-instances-autorun` executes `awsac-processjobs` with three files as parameters, containing all necessary information: the **job definitions file**, the **session information file** and the **instance ID file**.

3. **Selecting jobs and start processing**

   Consider one instance within the session. There `awsac-processjobs` parses the **job definitions file**, the **session information file** and the **instance ID file**. As described above, from this information `awsac-processjobs` is able to determine the jobs for this considered instance.

   The user must define an *EBS snapshot ID* (see *Elastic Block Store*) for each job. This makes it possible, to execute different jobs with different *ATLAS Release Versions*. The *snapshot IDs* are defined in the **job definitions file**, too. Hence, when `awsac-processjobs` knows which jobs have to be executed on this instance, it knows which different *EBS snapshots* have to be embedded into the system (in form of new *EBS volumes*). After detecting which *ATLAS Software Release snapshots* are needed for this instance, `awsac-processjobs` uses *boto* to create *EBS volumes* from these *snapshots*. It then attaches these volumes to the current instance and mounts them into the file system.

   Maybe this won't happen very often, but this feature makes it possible, that e.g. two different *EBS Volumes* with two different *ATLAS Release Versions* are available from one instance with two virtual cores, running two jobs that demanded two different *snapshot IDs*.

   Each job shell script will get its own sub process called by `awsac-processjobs`. Every sub process gets its own working directory.

   Every time the status of a job changes, a SimpleDB domain (the *session ID* is the name of this domain) for this session will be created/updated using *boto*. Currently, there are three possible states: **running** (sub process started), **saving** (sub process ended, uploading results to *S3*) and **finished**. This monitoring information can be retrieved using `awsac-session -check`. So the user is able to follow the status of all his jobs in the session.

   While processing the jobs, `stdout` and `stderr` of the job shell scripts are collected into log files. `stdout` and `stderr` of `awsac-processjobs` itself is written to console (console output of an instance is receivable using a special *EC2 API call*, e.g. with *Elasticfox*) and into a log file, too.

4. **Monitor status information**

   As stated above, `awsac-session -check` (together with the *session ID*) retrieves the status information set by all `awsac-processjobs` scripts on

the different instances within the session. It gives a convenient overview about states, returncodes, times (start, end, duration, ...) and more of all jobs.

5. **End processing jobs**

   Assume one special instance with a job shell script sub process just finished. The returncode of the script is written to *SimpleDB* and if the job shell script created a `results.tar.bz2` in its working directory, `awsac-processjobs` will save it to *S3*. The log file containing `stdout` and `stderr` of the script is compressed and uploaded, too. All these output/results can be received using `awsac-session -getresults`.

   When all the uploading has finished, the *EBS volumes* will be unmounted, detached and deleted by `awsac-processjobs`. As a last step the `awsac-processjobs` log file is bundled and uploaded to S3, too. Finally `awsac-processjobs` terminates the instance it is running on with an *EC2 API call*.

6. **Get results**

   By using `awsac-session -getresults` together with the *session ID*, the client will receive all the compressed archives of job results and log files.

7. **Clean up**

   By using `awsac-session -cleanup` together with a *session ID*, the client offers the possibility to delete contents from *S3* and/or *SimpleDB*.

6.3 How to use the AWSACtools job system

If you like to know how to use the job system step by step, you should read the next parts.

6.3.1 Build job session information files

To start a job session, the user has to call `awsac-session -start -ini sessionstartinfofile --archive sessionarchivefile`. So, the *session start info file* and the *session archive* are needed. This part will explain the meaning and content of these files.

session archive:

This file will be downloaded by each instance within the session. **It is intended to contain small, but essential information**. It should not be used to distribute big masses of input data. The *session archive* **must** include the following:

- `awsac-all-instances-autorun`
- `awsac-processjobs`
- job shell script(s)
- job definitions file (explained in the next paragraph)

Since the *session archive* is an easy way to make small amounts of data available to the instances, it may be convenient to include additional files into the *session archive*. This can be any script, config file, small data file or anything else small that is e.g. needed in any job shell script.

The archive itself **must** be a *BZ2* compressed tarball. Any other file extension than `.tar.bz2` will be declined by `awsac-session`.

job definitions file:

Until here, you perhaps asked yourself how to define the jobs a job session should process. The solution is the **job definitions file**. A job is defined by its shell script and the *snapshot ID* (see *Elastic Block Store*) containing the *ATLAS Software Release Version* the job should use. Hence, the data format of one line in the *job definitions file* is the following:

```
snapshotID;shellscriptname;numberofjobs
```

This means that one line can define more than one job. Hence, if you like to run different copies of the same job shell script with the same *ATLAS Software Release Version*, you only need one line. If you like to define jobs that differ, you have to write more than one line.

Consider, the shell script `shellscriptA.sh` should run three times using snapshot `snap1` and one time using `snap2`. Additionally, `shellscriptB.sh` should run two times using `snap1`, too. The following *job definitions file* will achieve this:

```
snap1;shellscriptA.sh;3
snap2;shellscriptA.sh;1
snap1;shellscriptB.sh;2
```

This file defines **six** jobs and should result in six simultaneously used virtual cores.

The *job definitions file* **must be named** `jobs.cfg` and placed **into the session archive***.

session start information file:

This file contains various information, `awsac-session -start` needs to start the job session. The information must be placed into an **ini-style** file. The following format **must** be matched:

```
[startinfo]
instance_type =
ami_id =
sessionsbucket =
ec2_uid =
shortdescr =
n_jobs =
```

- `instance_type` defines the favoured *EC2 instance type* type. Currently *m1.small* and *c1.medium* are supported. This implicates the number of *cores per instance* (`n_cpi`).
- `ami_id` is the ID of the *AWSAC AMI* to start up. The AWS account (defined by the credentials Access Key ID and Secret Key) you use with `awsac-session` must have access to this AMI (e.g. an own or a public AMI).
- `sessionbucket` is the *S3* bucket the session stores its *S3* objects in (the *session archive*, all the result files, ...) Your AWS account needs full access to this bucket.
- `ec2_uid` is the *EC2* User ID the AMI defined in `ami_id` belongs to. In case of a public AMI, this perhaps is not your UID. `awsac-session` needs that to check the existence of the AMI.
- `shortdescr` must be a *short* (not longer than 8 chars) description of the job session.
- `n_jobs` defines the number of jobs within the session. Hence, it defines the number of instances `n_inst` to start. `awsac-session` will calculate `n_inst` from `n_jobs` and `n_cpi`. As you can see, `n_jobs` should match the number of jobs defined in `jobs.cfg`.

6.3.2 Detailled awsac-session manual

This part is a detailed manual for using the job system with `awsac-session`. Some facts that were already explained before will be repeated.

What is a session? A *job session* is a bunch of jobs started with "one command". Technically this means, that one or more *EC2* instances are started within one *RunInstances API call*. So within one job session there may exist only one *EC2* reservation. The number of *EC2* instances `n_inst` in the job session depends on the number of jobs `n_jobs` and the number of *cores per instance* `n_cpi`.

What is a job? A job basically is a shell script completely written on your own. One job runs on one virtual core on one instance within a sub process of a *Python* script (`awsac-processjobs`). If you choose *instance type c1.medium* (with two virtual cores

per instance) for your session (all instances within one session are of the same type), then two jobs ("/bin/sh shellscript" sub processes) will run on each *EC2* instance. Of course, there may be one instance that is only occupied by one job. Your jobs are defined in the *job definitions file* in the *session archive*. There you must place the shell scripts, too. Other files the instances need, can be included, here as well. *Above*, you can find more details about the *session archive* and the *job definitions file*.

What does awsac-session do in detail? There are four main tasks: Start sessions (`-start` commandlineoption), check sessions' status (`-check`), get sessions results (`-getresults`) and clean up used *Amazon Web Services S3* and *SimpleDB* (`-cleanup`). These different options are now explained in detail.

- `-start -ini sessionstartinfofile --archive sessionarchivefile`: Starts a session.

  `awsac-session` needs some information about the session to start. This information must be delivered within the **session start info** and **session archive** files. How to build the content of these is described in *5.3.1 Build job session information files*.
  After checking the settings in the *session start info file*, `awsac-session` generates a *session ID* for the new session. It consists of the time, the short description and some randomness (it must be a unique identifier). Then the *session archive* is uploaded to *S3* (to `sessionbucket/sessionid/archivefilename`). The existence of the given AMI is checked. `awsac-session` builds a special **userdata string** (see *4.3.1: userdata string*) that will submitted within the *RunInstances API call* to start up the instances for the session. The string has the following format:

  ```
  session_id;sessionbucket;archivefilename;cores_per_instance
  ```

  Hence, when instances of an *AWSAC AMI* receive this string, they know the *session ID* of the session they belong to and are able to download the *session archive* from *S3*. Of course, an instance should know by itself how many cores it has. But by adding `n_cpi` to the string, it is possible to control the number of jobs per instance from the client system. `awsac-processjobs` assimilates this information and only starts as many sub processes per instance as given here.
  `awsac-session` checks the existence of a *SimpleDB* domain with the *session ID* as its name. If it exists, it will be deleted. All content will get lost. Then `awsac-session` asks the user, if he really wants to start `n_inst` instances of the defined type with the constructed *userdata string*. Choosing `y` causes `awsac-session` to send the *RunInstances API call*. On success, the *EC2* reservation ID is displayed.
  After this **a config file is written**. It contains all information about the session and is necessary for `awsac-session -check`, `-getresults` and `-cleanup`. The filename of the config file contains the *session ID*, leading

in unique names of the produced config files. Keeping these configfiles even after `-cleanup` means keeping information about a session.
Note: After the *RunInstances API call* mainly two different things can happen: All now automatically following steps result in success or, if there is a problem with e.g. the user data or the archive content, then all instances will fail. Some recommendations:

1. always check the instances with *Elasticfox*. There the states of the instances can be watched (*pending* or *running* or already *shutting-down*). *running* means that the operating system has started booting.
2. use the console output option in *Elasticfox*. It allows you to see the output of an instance. All server-sided steps should raise errors if something went wrong. These errors can be seen in the console output. The output is not delivered in realtime. There is some delay!
3. if there are no instances running at all (AWS credentials in *Elasticfox* must be the same `awsac-session` used), then perhaps the *RunInstances API call* failed. We never observed this behaviour.

- `-check -config configfile`: Checks SimpleDB for the state of a session.

  When `awsac-processjobs` runs with success until starting the jobs as sub processes, then the *SimpleDB* domain *session ID* has been created and filled with content. `awsac-session` gets this information from *SimpleDB* and prints it to the screen. The output should be self-explanatory. The *EC2* instance state switch from *pending* to *running* may last a few minutes. Since the server-sided *AWSACtools* components start after *init* from `rc.local` and then take some seconds for them self, there may decay some minutes from `-start` until the first job reports something to the *SimpleDB* domain of the session.
  Hence, when `awsac-session` states that the domain *session ID* cannot be found some minutes after `-start`, then this should be okay. A maximum delay of about 4 minutes was experienced. Everything above 5 minutes should raise doubts. If the domain does not exist after ~10 minutes, then something went wrong. Check console output of the instances with *Elasticfox*.
  If a job makes problems (e.g. it is running for a long time and should have finished long ago), then you should use *Elasticfox* to find out the *public DNS name* of the instance the problematic job is running on. Use *ssh* to connect to the instance. Then you have various possibilities to debug the problem. An example of what you can do:

  Look in one of the different log files and decide, whether the job should go on or should be killed. If necessary, kill the "/bin/sh shellscript" sub process that makes the problems. Then `awsac-processjobs` should go on processing the remaining steps until instance shut-down. This would be a clean solution, because other jobs on the same instance are not affected.

- `-getresults -config configfile -outdir outputdir`: Downloads all result files from the sessionbucket.

  Started with these parameters, `awsac-session` connects to *S3* and checks if the sessionbucket exists. If it exists, all objects in this bucket beginning with *session ID* are downloaded to a directory in `outputdir` with the *session ID* in its name.

- `-cleanup -config configfile`: Asks you for cleaning *S3* and/or *SimpleDB*.

  When all jobs are finished and all result files are downloaded, *S3* and *SimpleDB* should be cleaned up. If you choose to clean *S3*, then all `sessionbucket/sessionid*` objects will be deleted. `awsac-session` informs you about every deleted object. If you choose to clean *SimpleDB*, then simply the whole *SimpleDB* domain *session ID* gets deleted with all its content.

Note: `awsac-session` uses *boto*. *boto* gets the AWS credentials from environment variables. Make sure that the following ones are set when running `awsac-session`: `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`

6.3.3 An example run - event generation

This example shows the generation of *single pions* within four jobs on two *c1.medium* instances. The jobs use *ATLAS Software Release* `14.2.10`, which is prepared in snapshot `snap-9fd433f6`. Two jobs should produce 30 pions, the other two jobs should produce 1000 pions. The `.root` files should be returned.

Preparation:

Okay, lets build the **job shell scripts**, the **jobs definitions file**, the **session archive** and the **session start information file**!

The two different *job shell scripts* will be named `gen_30.sh` and `gen_1000.sh`. Hence, the corresponding *job definitions file*, `jobs.cfg`, has the following content:

```
snap-9fd433f6;gen_30.sh;2
snap-9fd433f6;gen_1000.sh;2
```

This defines the **four** desired jobs. The first two will do the same and the last two will do the same, too.

`gen_1000.sh` is presented here:

```
echo ATLAS JOB BEGIN: gen 1000 events
echo $(date)
echo ==========================================

source ${ATLASDir}/14.2.10/cmthome/setup.sh -tag=14.2.10
```

```
csc_evgen_trf.py 007410 1 1000 765432
   ${AWSACworkingDir}/CSC.007410.singlepart_singlepi+_logE.py
   EVGEN_007410_00001.pool.root > csc_evgen_trf_gen1000.log
echo ============================================

echo BUNDLE RESULTFILES INTO ARCHIVE
tar cjvf results.tar.bz2 EVGEN_007410_00001.pool.root csc_evgen_trf_gen1000.log
if [ $? -eq 0 ]; then
    echo "bundle successfull"
else
    echo "tar error"
fi

echo ============================================
echo $(date)
echo ATLAS JOB END: gen 1000 events
exit
```

This is the content of `gen_30.sh` (for completeness):

```
echo ATLAS JOB BEGIN: gen 30 events
echo $(date)
echo ============================================

source ${ATLASDir}/14.2.10/cmthome/setup.sh -tag=14.2.10
csc_evgen_trf.py 007410 1 30 765432
   ${AWSACworkingDir}/CSC.007410.singlepart_singlepi+_logE.py
   EVGEN_007410_00001.pool.root > csc_evgen_trf_gen30.log
echo ============================================

echo BUNDLE RESULTFILES INTO ARCHIVE
tar cjvf results.tar.bz2 EVGEN_007410_00001.pool.root csc_evgen_trf_gen30.log
if [ $? -eq 0 ]; then
    echo "bundle successfull"
else
    echo "tar error"
fi

echo ============================================
echo $(date)
echo ATLAS JOB END: gen 30 events
exit
```

As you can see, at the beginning of every *job shell script* that like to use *ATLAS Software* commands, the *ATLAS Software Release* must be initialized. Details about this initialization can be found in *7.2 Install and configure ATLAS Software Release*.

Details on particle generation will be delivered in `CSC.007410.singlepart_singlepi+_logE.py`. This file will be stored in the *session archive*. So it will be accessible from the *job shell scripts* at runtime!

`awsac-autorun` sets and uses some environment variables that can be used within the *job shell scripts*, too. The environment variable `AWSACworkingDir` contains the directory, where `awsac-autorun` stores the contents of the *session archive*; `ATLASDir` contains the path, where one or more *ATLAS Software Release* versions are mounted to by `awsac-processjobs`. The files `results.tar.bz2` produced by both of the *job shell scripts* will be collected and sorted by the job system automatically.

From this it follows that we already have various files for the *session archive*: `gen_1000.sh`, `gen_30.sh`, `jobs.cfg` and `CSC.007410.singlepart_singlepi+_logE.py`. Together with `awsac-processjobs` and `awsac-all-instances-autorun` the content will be complete. Hence, additionally `awsac-processjobs.py` and `awsac_all_instances_autorun.sh` were included (These are the real file names of the programs at the time of creating this example). The files were bundled into a *BZ2* compressed tarball `sessionarchive.tar.bz2`. This may help you to create the archive:

```
$ cd directory_of_session_archive_files
$ tar cjvf sessionarchive.tar.bz2 *
```

Additionally, this little *Python* script may help you with that, too (e.g. convenient on *Windows* systems):

```
import tarfile, os

bundledir = "directory_of_session_archive_files"

tar = tarfile.open("sessionarchive.tar.bz2", "w:bz2")
for file in os.listdir(bundledir):
    tar.add(os.path.join(bundledir,file),file)
tar.close()
```

This is not all information `awsac-session` needs to start a new *job session*, yet. The *session start info file* is still missing: `session.ini`. In the case of the example, the content of `session.ini` looks like:

```
[startinfo]
instance_type = c1.medium
ami_id = ami-c97591a0
sessionsbucket = atlassessions
ec2_uid = 201521871620
shortdescr = eventgen
n_jobs = 4
```

We want to have **«four» jobs within the session** and they should run on **«c1.medium» instances** of the **AWSAC AMI «ami-c97591a0»**; owned by **AWS user «201521871620»**. `awsac-session` and `awsac-processjobs` will work with the **bucket «atlassessions»**. This example **session is called «eventgen»**.

Start session:

Ready to go! Type

```
$ ./awsac-session.py --start -i session.ini -a sessionarchive.tar.bz2
```

The output should look like

```
::::>  AWSACtools session management v08-10-13
::::>  by Jan-Philip Gehrcke


# starting new session; parsing start information file...

[startinfo]
sessionsbucket = atlassessions
ami_id = ami-c97591a0
shortdescr = eventgen
ec2_uid = 201521871620
instance_type = c1.medium
n_jobs = 4

# ----- shortdescription of new session: EVENTGEN -----

# instancetype chosen: ‘c1.medium‘ with 2 core(s) per instance.
   With 4 demanded job(s)/core(s) this makes 2 instance(s) we have to start.
   Unused cores: 0.

# generating session ID from date, shortdescription, randomness:
081215_1646--eventgen--1816

# uploading sessionarchive file to S3...
uploaded to S3 as object (bucket:‘atlassessions‘
   key:‘081215_1646--eventgen--1816/sessionarchive.tar.bz2‘).

# connecting to EC2 to check your AMI-ID...
found AMI ‘ami-c97591a0‘ (ATLAS/SL47-AWSAC-v03-boto957.manifest.xml)

# building user-data string to be submitted to all instances...
081215_1646--eventgen--1816;atlassessions;sessionarchive.tar.bz2;2
(session_id;bucket;sessionarchivename;cores_per_instance)
```

```
# checking SimpleDB domain for this session...
  domain does not exist: 081215_1646--eventgen--1816

# In the next step EC2 will be instructed to run exactly 2 instance(s)
   of given AMI (type 'c1.medium') with user-data mentioned obove.
proceed? (y/n):
```

If you now choose `y`, `awsac-session` goes on and the *EC2* reservation will be requested. The output then looks like:

```
Request accepted. EC2 reservation ID: r-bc0da0d5
# saving session to file... session-081215_1646--eventgen--1816.cfg
```

As you can see, a new file was created: `session-081215_1646-eventgen-1816.cfg`.

Monitor session:

Use *Elasticfox* and `awsac-session -check` to follow the sessions process. Watch the instances pending...

Reservation ID	Owner	Instance ID	AMI	AKI	...	State
r-bc0da0d5	201521871620	i-33f64f5a	ami-c97591a0			pending
r-bc0da0d5	201521871620	i-32f64f5b	ami-c97591a0			pending

...and starting...:

Reservation ID	Owner	Instance ID	AMI	AKI	...	State
r-bc0da0d5	201521871620	i-33f64f5a	ami-c97591a0			running
r-bc0da0d5	201521871620	i-32f64f5b	ami-c97591a0			running

After some time, you can see new attached *EBS volumes* of the desired snapshot listed:

VOL ID	Siz...	SNAP ID	Availability Zone	Status	Local Creation Date	Instance ...	Device	Attachment...
vol-92e400fb	20	snap-9fd433f6	us-east-1b	in-use	2008-12-15 16:48:42	i-33f64f5a	/dev/sdh1	attached
vol-95e400fc	20	snap-9fd433f6	us-east-1b	in-use	2008-12-15 16:49:20	i-32f64f5b	/dev/sdh1	attached

This indicates that the *chain of commands* was successful until `awsac-processjobs`. Hence, there already should be some monitoring information:

```
$ ./awsac-session.py --check -c session-081215_1646--eventgen--1816.cfg
```

The output is the follwing:

```
::::>  AWSACtools session management v08-10-13
::::>  by Jan-Philip Gehrcke
```

```
[sessionconfig]
n_jobs = 4
cores_per_instance = 2
shortdescr = eventgen
bucket = atlassessions
session_id = 081215_1646--eventgen--1816
ec2_reservation_id = r-bc0da0d5
instance_type = c1.medium
unused_cores = 0
n_instances = 2
ec2_uid = 201521871620
ami_id = ami-c97591a0
runuserdata = 081215_1646--eventgen--1816;atlassessions;sessionarchive.tar.bz2;2

============= Job 1 (status: finished) ==============
running /mnt/awsac/gen_30.sh on instance i-33f64f5a with launchindex 0
started running: 08-12-15 16:49:03
ended running: 08-12-15 16:50
returncode: 0
started saving: 08-12-15 16:50:33
ended saving: 08-12-15 16:50:39

============= Job 2 (status: saving) ==============
running /mnt/awsac/gen_30.sh on instance i-33f64f5a with launchindex 0
started running: 08-12-15 16:49:08
ended running: 08-12-15 16:50
returncode: 0
started saving: 08-12-15 16:50:51

============= Job 3 (status: running) ==============
running /mnt/awsac/gen_1000.sh on instance i-32f64f5b with launchindex 1
started running: 08-12-15 16:49:39

============= Job 4 (status: running) ==============
running /mnt/awsac/gen_1000.sh on instance i-32f64f5b with launchindex 1
started running: 08-12-15 16:49:44
```

The jobs with 30 event generations are a bit faster than the ones generating 1000 events.

When the jobs are over, then the *EBS volumes* should be listed as *deleting* or *deleted*:

VOL ID	Siz...	SNAP ID	Availability Zone	Status	Local Creation Date	Instance ...	Device	Attachment...
vol-92e400fb	20	snap-9fd433f6	us-east-1b	deleting	2008-12-15 16:48:42			
vol-95e400fc	20	snap-9fd433f6	us-east-1b	deleting	2008-12-15 16:49:20			

Of course, `./awsac-session.py -check -c session-081215_1646-eventgen-1816.cfg` now confirms that all jobs are **finished**.

Get results:

Lets download the content of the *sessionbucket*:

```
$ mkdir sessionresults
$ ./awsac-session.py --getresults -c session-081215_1646--eventgen--1816.cfg
   -o sessionresults
```

The output is the following:

```
::::>  AWSACtools session management v08-10-13
::::>  by Jan-Philip Gehrcke


[sessionconfig]
n_jobs = 4
cores_per_instance = 2
shortdescr = eventgen
bucket = atlassessions
session_id = 081215_1646--eventgen--1816
ec2_reservation_id = r-bc0da0d5
instance_type = c1.medium
unused_cores = 0
n_instances = 2
ec2_uid = 201521871620
ami_id = ami-c97591a0
runuserdata = 081215_1646--eventgen--1816;atlassessions;sessionarchive.tar.bz2;2

outputfolder created: /home/iwsatlas1/gehrcke/dokusession/
   sessionresults/session-081215_1646--eventgen--1816
saved processjobslog_LI_0.tar.bz2
saved processjobslog_LI_1.tar.bz2
saved results_job_1.tar.bz2
saved results_job_2.tar.bz2
saved results_job_3.tar.bz2
saved results_job_4.tar.bz2
saved sessionarchive.tar.bz2
saved stdouterr_job_1.tar.bz2
saved stdouterr_job_2.tar.bz2
saved stdouterr_job_3.tar.bz2
saved stdouterr_job_4.tar.bz2
```

The numbering of the jobs follows the order of the definitions in the **job definitions file** `jobs.cfg`. The `awsac-processjobs` log exists for each instance within the session; labelled with the *launch index* (LI) in the file names. All the extracted result- and log archives above can be found here: http://gehrcke.de/awsac/permstuff/example_session/results

> **Warning:** Since the *ATLAS Software* prints the current environment, the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are sent to stdout, too. These keys were removed from the log files.

Clean up:

The monitoring information on *SimpleDB* is no longer needed and the results and logs were downloaded from *S3*.

```
$ ./awsac-session.py --cleanup -c session-081215_1646--eventgen--1816.cfg
```

Confirm to delete the contents. Then the output looks like:

```
::::>  AWSACtools session management v08-10-13
::::>  by Jan-Philip Gehrcke


[sessionconfig]
n_jobs = 4
cores_per_instance = 2
shortdescr = eventgen
bucket = atlassessions
session_id = 081215_1646--eventgen--1816
ec2_reservation_id = r-bc0da0d5
instance_type = c1.medium
unused_cores = 0
n_instances = 2
ec2_uid = 201521871620
ami_id = ami-c97591a0
runuserdata = 081215_1646--eventgen--1816;atlassessions;sessionarchive.tar.bz2;2

delete SimpleDB-domain 081215_1646--eventgen--1816? y/n: y
  domain existed and was deleted: 081215_1646--eventgen--1816
delete S3-objects atlassessions/081215_1646--eventgen--1816/* ? y/n: y
deleted processjobslog_LI_0.tar.bz2
deleted processjobslog_LI_1.tar.bz2
deleted results_job_1.tar.bz2
deleted results_job_2.tar.bz2
deleted results_job_3.tar.bz2
deleted results_job_4.tar.bz2
deleted sessionarchive.tar.bz2
deleted stdouterr_job_1.tar.bz2
deleted stdouterr_job_2.tar.bz2
deleted stdouterr_job_3.tar.bz2
deleted stdouterr_job_4.tar.bz2
```

6.3.4 The job system from the instances point of view

This part has the intention to make the job system more comprehensible by means of the *console output* (the log) of an instance within the *job session* above. The *console output* logfiles of both instances within the example session can be found here: http://gehrcke.de/awsac/permstuff/example_session/consoleoutput

Now let the *console output* of the instance with *launch index* `0` explain the job system step by step.

At first the instance starts up:

```
Linux version 2.6.16-xenU [...]
[...]
INIT: Entering runlevel: 4
[...]
Starting HAL daemon: [  OK  ]
```

When `INIT` finishes, `awsac-autorun` starts up (invoked by `rc.local`):

```
************************************************************
  AWSACtools: autorun (Individual Instance Startup System)
    /root/awsac/awsac-autorun.sh invoked from rc.local
                  by Jan-Philip Gehrcke
                      v08-10-13
************************************************************
# setting up environment variables...
ATLASDir: /mnt/atlas
ATLASworkingDir: /mnt/atlasworkarea
AWSACworkingDir: /mnt/awsac
SessionInfoDir: /mnt/awsac/sessioninfo
SessionInfoFile: /mnt/awsac/sessioninfo/awsac.sessinfo
SessionArchiveFile: /mnt/awsac/sessionarchive.tar.bz2
AWSACAutostartFile: /mnt/awsac/awsac_all_instances_autorun.sh
# creating directories for AWSAC and ATLAS-Software in /mnt...
creating AWSACworkingDir
creating SessionInfoDir
creating ATLASDir
creating ATLASworkingDir
```

After creating directories and environment variables, `awsac-autorun` tries to receive the *userdata string*:

```
# getting user-data (should contain the sessioninfostring)...
--16:48:41--  http://169.254.169.254/latest/user-data
           => '/mnt/awsac/sessioninfo/awsac.sessinfo'
Connecting to 169.254.169.254:80... connected.
HTTP request sent, awaiting response... 200 OK
```

```
Length: 66 [application/octet-stream]
 0% [                                     ] 0     --.--K/s
   100%[====================================>] 66   --.--K/s
16:48:41 (8.99 MB/s) - '/mnt/awsac/sessioninfo/awsac.sessinfo' saved [66/66]
```

On success, the *Python* script `getsessionarchive` uses the string information to download the *session archive*:

```
# setting AWSenvironment variables...
# running getsessionarchive -i /mnt/awsac/sessioninfo/awsac.sessinfo
   -o /mnt/awsac/sessionarchive.tar.bz2
# to get sessionarchive from S3 bucket...
getsessionarchive.py: downloaded 081215_1646--eventgen--1816/sessionarchive.tar.bz2
   from bucket atlassessions to /mnt/awsac/sessionarchive.tar.bz2
# untaring sessionarchive...
awsac-processjobs.py
awsac_all_instances_autorun.sh
CSC.007410.singlepart_singlepi+_logE.py
gen_1000.sh
gen_30.sh
jobs.cfg
# trying to execute AWSAC autostart shellscript...
```

After extracting the archive, `awsac-all-instances-autorun` is executed:

```
******* awsac_all_instances_autorun.sh *******
*******           v08-12-14            *******
# getting instance-id (from meta-data server)...
--16:48:42--  http://169.254.169.254/latest/meta-data/instance-id
          => '/mnt/awsac/sessioninfo/InstanceID'
Connecting to 169.254.169.254:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 10 [text/plain]
 0% [                                     ] 0     --.--K/s
   100%[====================================>] 10   --.--K/s
16:48:42 (1.59 MB/s) - '/mnt/awsac/sessioninfo/InstanceID' saved [10/10]
instance id: i-33f64f5a
running /opt/bin/python /mnt/awsac/awsac-processjobs.py
   --sessioninfofile /mnt/awsac/sessioninfo/awsac.sessinfo
   --jobsfile /mnt/awsac/jobs.cfg --instanceID i-33f64f5a
```

It received the *instance ID* and initializes `awsac-processjobs` with the proper *session info* and *job definitions* files. `awsac-processjobs` starts up.

```
****** processjobs.py start Mon Dec 15 16:48:42 2008 ******
# parsing the sessioninfofile /mnt/awsac/sessioninfo/awsac.sessinfo ...
  Session ID: 081215_1646--eventgen--1816 ;  CoresPerInstance: 2
```

```
# getting information about this instance from EC2...
  the ami-launch-index of this instance (i-33f64f5a) is 0
  the Availability Zone of this instance is us-east-1b
# parsing the jobsconfigfile /mnt/awsac/jobs.cfg ...
# jobsdatadicts_list read from /mnt/awsac/jobs.cfg
# my job numbers are: [1, 2]
[{'jobnr': '1', 'snap_id': 'snap-9fd433f6', 'shscript': '/mnt/awsac/gen_30.sh'},
    {'jobnr': '2', 'snap_id': 'snap-9fd433f6', 'shscript': '/mnt/awsac/gen_30.sh'}]
# prepare the creation of EBS volume(s) from the needed ATLAS Release snapshot(s)...
  detected following different snap_ids: ['snap-9fd433f6']
  planing to assign snap_id(s) to following device(s):
  snap-9fd433f6->/dev/sdh1
```

awsac-processjobs identified the jobs it has to run (from jobs.cfg and the *launch index*). It built a list of needed snapshots for *EBS volumes* and assigned corresponding devices. The *EBS Volume(s)* will be mounted in the same *availability zone* as the instance.

```
# create EBS volume(s)...
  instructed EC2 to create EBS from snapshot snap-9fd433f6
  status: creating
  status: available
# attach EBS volume(s)...
  instructed EC2 to attach volume vol-92e400fb to /dev/sdh1
  status: attaching
  status: attached
# mount EBS volume(s)...
  invoke mounting: subprocess.Popen() with args ['mount', '/dev/sdh1', '/mnt/atlas']
  mount subprocess ended. returncode: 0
# this is the list of successfully mounted EBS - snapshots:
  ['snap-9fd433f6']
# the following jobs now have their desired EBS running in system:
[{'jobnr': '1', 'snap_id': 'snap-9fd433f6', 'shscript': '/mnt/awsac/gen_30.sh'},
   {'jobnr': '2', 'snap_id': 'snap-9fd433f6', 'shscript': '/mnt/awsac/gen_30.sh'}]
```

awsac-processjobs has guaranteed a successfully mounted snapshot for its jobs 1 and 2 (for all jobs it has to run). So the jobs can start working.

```
# initialize running jobs...
  creating SimpleDB jobitems...
# preparing job 1...
  cwd for job: /mnt/atlasworkarea/1  (created)
  calling subprocess.Popen() with args: ['/bin/sh', '/mnt/awsac/gen_30.sh']
  runstart: updating sDB item job1
# preparing job 2...
  cwd for job: /mnt/atlasworkarea/2  (created)
  calling subprocess.Popen() with args: ['/bin/sh', '/mnt/awsac/gen_30.sh']
```

```
  runstart: updating sDB item job2
# wait for subprocesses to finish...
```

The *job shell script* subprocesses are started and the first monitoring information is written to *SimpleDB*. `awsac-processjobs` now is within a waiting loop.

```
# subprocess for job 1 ended.  returncode: 0
  runend: updating sDB item job1
  savestart: updating sDB item job1
  found /mnt/atlasworkarea/1/results.tar.bz2 (47507 Byte)
08-12-15 16:50:34:  start upload. bucket:atlassessions;
   key:081215_1646--eventgen--1816/results_job_1.tar.bz2
08-12-15 16:50:34:  finished.
  outputfile of job 1 found: /mnt/atlasworkarea/1/stdouterr_job_1.log
  bundled /mnt/atlasworkarea/1/stdouterr_job_1.log to
   /mnt/atlasworkarea/1/stdouterr_job_1.tar.bz2
  planing to upload /mnt/atlasworkarea/1/stdouterr_job_1.tar.bz2 (314 Byte)...
08-12-15 16:50:34:  start upload. bucket:atlassessions;
   key:081215_1646--eventgen--1816/stdouterr_job_1.tar.bz2
08-12-15 16:50:34:  finished.
  saveend: updating sDB item job1
```

The first job finished. Monitoring information is updated. Result and log (compressed) is saved to *S3*.

```
# subprocess for job 2 ended.  returncode: 0
  runend: updating sDB item job2
  savestart: updating sDB item job2
  found /mnt/atlasworkarea/2/results.tar.bz2 (47574 Byte)
08-12-15 16:50:51:  start upload. bucket:atlassessions;
   key:081215_1646--eventgen--1816/results_job_2.tar.bz2
08-12-15 16:50:51:  finished.
  outputfile of job 2 found: /mnt/atlasworkarea/2/stdouterr_job_2.log
  bundled /mnt/atlasworkarea/2/stdouterr_job_2.log to
   /mnt/atlasworkarea/2/stdouterr_job_2.tar.bz2
  planing to upload /mnt/atlasworkarea/2/stdouterr_job_2.tar.bz2 (314 Byte)...
08-12-15 16:50:51:  start upload. bucket:atlassessions;
   key:081215_1646--eventgen--1816/stdouterr_job_2.tar.bz2
08-12-15 16:50:52:  finished.
  saveend: updating sDB item job2
```

The same with the second job. No more jobs left to wait for.

```
# all subprocesses ended
# summary:
Jobnumber: 1
starttime: 08-12-15 16:48:56
```

```
endtime: 08-12-15 16:50
executiontime: 00:01
returncode: 0
----------------------
Jobnumber: 2
starttime: 08-12-15 16:49:03
endtime: 08-12-15 16:50
executiontime: 00:01
returncode: 0
----------------------
```

This little summary informs about the finished sub processes. `awsac-processjobs` successfully ran the jobs. Now it has to clean up!

```
# unmount EBS volume(s)...
  invoke unmounting: subprocess.Popen() with args ['umount', '/dev/sdh1']
  umount subprocess ended. returncode: 0
# detach EBS volume(s)...
  instructed EC2 to detach volume vol-92e400fb
  status: detaching
  status: available
# delete EBS volume(s)...
  instructed EC2 to delete volume vol-92e400fb
  status: True
# close processjobs-logfile...
  it will then be bundled and uploaded. then EC2 will be instructed
   to terminate this instance
  bundled processjobs.log to processjobslog_LI_0.tar.bz2
08-12-15 16:51:02:  start upload. bucket:atlassessions;
   key:081215_1646--eventgen--1816/processjobslog_LI_0.tar.bz2
08-12-15 16:51:02:  finished.
# instructing EC2 to terminate my instance..
```

As you can see, `awsac-processjobs` leaves nothing on *EC2*. The last few lines can not be seen in `processjobs.log`. In the following last part of the *console output*, you can see the shut-down of the instance:

```
c
Scientific Linux SL release 4.7 (Beryllium)
Kernel 2.6.16-xenU on an i686
domU-12-31-39-01-C9-53 login: INIT: Switching to runlevel: 0
INIT: Sending processes the TERM signal
[...]
Halting system...
md: stopping all md devices.
md: md0 switched to read-only mode.
System halted.
```

This instance had a successful run within its job session and terminated itself successfully!

CHAPTER

SEVEN

CREATING AN AMI FOR AWSAC FROM SCRATCH

In this chapter it is shown, how to create an **AWSAC AMI**: an *Amazon Machine Image (AMI)* for *ATLAS Computing*, as it is used by the job system described in the *chapter before*. The whole process is described in every detail and from scratch; in order to afford easy reproduction.

The AMI will contain a specialized Linux system that fulfils the following requirements:

- running properly on *EC2*
- *ATLAS Software Release* runs properly on it

Note: At this point, we take the opportunity to refer to http://cernvm.cern.ch - a project with the aim to «provide a baseline Virtual Software Appliance for use by LHC experiments at CERN». Perhaps this will be important for the *AWSAC* project in the future.

7.1 Prerequisites

At first the operating system to run on *EC2* has to be chosen, just like the way to create the AMI. The decisions I made will be discussed shortly.

7.1.1 Choosing Scientific Linux 4

In general *EC2* is compatible to any Linux distribution. But to make things easy, it is recommended to use only new distributions with new software versions. This will arise the fewest problems. One reason for this is, that most tools that are needed to control *EC2* need recent software versions. There are more reasons and you will see the emerging problems by reading this documentation. It is important to know, that *EC2* is guaranteed to run properly with new *Fedora* systems. On the other hand *ATLAS Software* is only guaranteed to run properly on *Scientific Linux 4*. With this old software one has to expect problems

with *EC2*. It seemed that nobody else tried to get it running on *EC2*, so I could not expect any support.

Hence I tried to get *ATLAS Software* running properly on *Scientific Linux 5* and *Fedora 7, 8, 9*. These tries exposed that the effort to obtain a running *ATLAS Software Release* under these distributions is really big. I aborted these approaches and decided to get *Scientific Linux 4* running on *EC2*, since **«ATLAS Software is running properly only on Scientific Linux 4»** seemed to be the strongest condition.

7.1.2 Two different ways to create an AMI

Amazon EC2 AMI Tools provide two commands to create an Amazon Machine Image.

The first command `ec2-bundle-image` creates an AMI through a loopback file. So you have to push the system you want to run on *EC2* into an image file step by step from another running system on your home PC.

The second command `ec2-bundle-vol` creates an AMI from a running system. Using this you can e.g. set up the system you want to run on *EC2* as a virutal machine on your home PC. Then you edit and configure this system "from inside" (the conventional way). When you think that it is ready to run on *EC2*, you can bundle it to an AMI with `ec2-bundle-vol` at runtime. For this the system itself needs the *AMI Tools* installed.

The second possibility is more convenient for our purposes. So I decided to set up a virtual machine (VM) with *Scientific Linux* using VMware Player. In the following I assume that *VMware Player* is successfully installed on your system.

7.2 Set up a Scientific Linux VM for an AMI

In this part I will show you the way to a new *Scientific Linux (SL)* VM that is ready for an *Amazon Machine Image*. The process is described by means of *VMplayer* for virtualization and *32 bit SL 4.7* as linux distribution.

7.2.1 Preparation

At first create a directory to put all needed files in, including the virtual disk for the *Scientific Linux* system. For this reason it is necessary that there is some free disk space (at least three times the space required by the *Scientific Linux* system we plan to set up - 10 GB should be enough). In the following I assume that this new directory is `/SL47ami`.

There are mainly two options to get the *SL* packages you need during your *SL* installation. If you download the CD iso files, you can install *SL* from these local files. But if you want to do this, every iso file needs to be attached to the VM as IDE drive. I decided to install the new operating system using the online repository directly. This is more convenient than

the many-iso-files-way. But, of course, we need a small iso file here, too. Download the *SL installer* image (~6 MB) to the new folder:

```
$ wget http://ftp.scientificlinux.org/linux/scientific/47/i386/images/SL/boot.iso
```

A *VMware* virtual machines' hard disk drive is reading from and writing to a special file in the host filesystem. This file must be *vmdk* formatted. The open source machine emulator and virtualizer QEMU brings along *qemu-img* that can create files of a specific size and format these files as *vmdk*. After downloading and installing QEMU, you can use *qemu-img* to create a new *vmdk* file. For our purposes a virtual harddisk drive with 10 GB storage is big enough:

```
$ qemu-img create -f vmdk /SL47ami/SL47ami.vmdk 10G
```

Note: The size of the so-called instance storage on a running *EC2* instance is not affected by this choice later on. At this juncture it is sufficient to ensure that there is enough space for *Scientific Linux* itself.

If you want to start a virtual machine with *VMware Player*, you have to give a configuration file - a so-called *vmx* file - to the player. Create a new configuration file; e.g. `SL47ami.vmx`. In this file - in essence - the virtual hardware must be configured. For our purposes the most important hardware the VM needs is a cdrom-image drive with the *SL installer* "inside", a hard disk drive (corresponding to the created *vmdk* file) and an ethernet adapter. So something like the following lines should be put into the new configuration file (working for me):

```
config.version = "8"
virtualHW.version = "4"

displayName = "Scientific Linux 4.7 - minimal for AMI"
guestOS = "rhel4"

memsize = "1024"

floppy0.present = "FALSE"

ide0:0.present = "TRUE"
ide0:0.filename = "/mnt/scratch/gehrcke/virtual-disks/SL47ami.vmdk"

ide1:0.present = "TRUE"
ide1:0.deviceType = "cdrom-image"
ide1:0.startConnected = "TRUE"
ide1:0.fileName = "/home/iwsatlas1/gehrcke/virtual_SL47ami/boot.iso"

ethernet0.present = "TRUE"
ethernet0.connectionType = "nat"
```

```
ethernet0.addressType = "generated"
ethernet0.generatedAddress = "00:0c:29:fa:b6:cb"
ethernet0.generatedAddressOffset = "0"
```

A large part of this should be self-explanatory. `guestOS` describes the operating system class you want to run on your VM. In this case this is *Red Hat Enterprise Linux 4*. You should adjust `memsize` (in MB) of your VM to your real hardware and to the needs of the applications running on it. If you want to run different VMs at the same time, you should vary the `generatedAddressOffset`. This avoids equaling MAC addresses.

Now the VM is ready to start up:

```
$ vmplayer SL47ami.vmx
```

Note: The VM boots up like a normal computer. Since the hard disk drive is still empty, the virtual BIOS looks for a bootable disk in the cdrom drive. In this case then the *SL installer* boots up!

7.2.2 Installing Scientific Linux

The *SL installer* needs to know the installation method. I chose **linux text** installation. The installation menu is self-explanatory. Nevertheless I will mention every step because some settings are really important or a bit tricky.

At first choose your **language**.

In the next step you can decide whether you want to install from an http/ftp online repository or e.g. from local iso files. As I described above, I chose the online repository. http is convenient, so choose **http**.

Now you have to set up TCP/IP. Try choosing **DHCP config**. For me this sometimes resulted in endless waiting. The reason for this is not clear to me; I guess that there are problems with the *VMware* DHCP server managing the "virtual subnet" for virtual machines on the host system. If you encounter the same problem you can easily workaround: check out the `dhcpd.conf` file corresponding to the *VMnet* that should provide the internet connection. For me this was `/etc/vmware/vmnet8/dhcpd/dhcpd.conf` There you can get all information you need to configure TCP/IP manually. This was fast for me, even if *DHCP config* did not work properly.

In the **http setup** menu you have to submit the *SL* location on an http server omitting "http://". Enter the following:

```
website name: ftp.scientificlinux.org
Scientific Linux directory: linux/scientific/47/i386
```

You like to configure the installation on your own, so use the **Custom Installation Type**.

You also like partitioning the hard disk on your own. Choose **Disk Druid**. And **yes**, you know that all data will be lost!

Now **Disk Druid** wants to know what to do. The configuration I chose is described schematically in this summary:

```
Add Partition:
    mount point: /
    ext 3
    9500 MB
    force primary

Add Partition:
    swap
    fill all available space
    force primary
```

In the following steps I specified to **use Grub**, **not to pass any boot options** to the kernel and **not to use a grub password**. The *Boot Loader Configuration* was **OK**. I chose to **install the boot loader to the MBR** and **acknowledged IP and Hostname** configuration.

In terms of security I chose **not to use a firewall** and to **disable SELinux**. I think that those things are not necessary for our extremely specialized *EC2* instances and may produce problems. The EC2 Network Security regulation should care for a sufficient amount of security. This was a fast decision and maybe I am wrong.

Now decide on **additional Languages**, the **Time Zone** and the **Root Password**.

In the following **Package Group Selection** you have to select the components your new system should consist of. Select what you need. My minimal config is the following:

```
Main Tree:
    select:
        YUM
        APT
        Development Tools:
            select:
                all gcc-options
            deselect:
                Emacs
        Administration Tools:
            select:
                all
        System Tools:
            select:
                all
    deselect:
        everything else (really! we don't need any graphics etc.)
```

Note: Selecting all *gcc options* ensures that the compiling *ATLAS Software Release* applications run correctly (including *KitValidation MooEvent*).

Begin installation and **reboot** when it is done.

Done! Your new *Scientific Linux* VM is up and running. You can now log in as root.

As a first action do a **YUM update** to get the latest security updates:

```
$ yum update
```

7.2.3 Configure the system for EC2

The next big objective is to bundle and upload an AMI of our new system. For this you have to breach some hurdles. At first you have to implement the *Amazon EC2 AMI Tools* into the new system. The tools provide the command `ec2-bundle-vol` we want to use to bundle the running system into an AMI. And they provide the possibility to upload the AMI to Amazon's *Simple Storage S3* using `ec2-upload-bundle`. Get the *AMI Tools* running on an old system like *Scientific Linux 4* requires some new software. The *Amazon EC2 AMI Tools* do not provide the possibility to register an AMI on EC2. So you have to install and configure the *Amazon EC2 Command-Line Tools* (also called *API Tools*), too. But this is not everything you have to configure: the system's hardware detection tool *kudzu* needs some modification, too, so that the system is able to boot up correctly on *EC2*.

Note: If you download the *AMI Tools* as RPM and try to install it with `rpm -i ec2-ami-tools.noarch.rpm`, you will notice that some requirements/dependencies are not met by *Scientific Linux 4*. I needed a newer version of *tar* (greater or equal to 1.15) and a newer version of *Ruby* (greater or equal to 1.8.2). After some investigation I decided to install both of them from source and then to install the RPM without regarding dependencies (see below). Maybe going around the package manager (and perhaps SRPMs) is not the cleanest way but it is easy and it really works good, as you can see in the following parts.

Install a newer tar: Download the latest source tarball, extract it and `cd` to the source directory. This will look like

```
$ wget http://ftp.gnu.org/gnu/tar/tar-1.20.tar.bz2
$ tar xjf tar-1.20.tar.bz2
$ cd tar-1.20
```

We want to brutally overwrite the the original *tar*. The executable's path is `/bin/tar`. So start `configure` with `-prefix=/`, compile with `make` and then install the new *tar*:

```
$ ./configure --prefix=/
$ make
$ make install
```

`/bin/tar` should now be the newer version. Test it:

```
$ which tar
$ tar --version
```

Install Ruby: Download the latest source tarball, extract it and `cd` to the source directory. This will look like

```
$ wget ftp://ftp.ruby-lang.org/pub/ruby/1.8/ruby-1.8.7.tar.bz2
$ tar xjf ruby-1.8.7.tar.bz2
$ cd ruby-1.8.7
```

We want to place the executable in `/usr/bin`. So start `configure` with `-prefix=/usr`, compile with `make` and then install *Ruby*:

```
$ ./configure --prefix=/usr
$ make
$ make install
```

`/usr/bin/ruby` should now exist. Check it out and test your new *Ruby* installation:

```
$ which ruby
$ ruby --version
```

Install the AMI Tools: Download and install the RPM without regarding dependencies:

```
$ wget http://s3.amazonaws.com/ec2-downloads/ec2-ami-tools.noarch.rpm
$ rpm -i --nodeps ec2-ami-tools.noarch.rpm
```

Test it! Type `ec2-` and then press TAB to see the new commands available. Try e.g.

```
$ ec2-upload-bundle --help
```

The following error should be normal:

```
/usr/lib/site_ruby/aes/amiutil/uploadbundle.rb:1:in ‘require’:
no such file to load -- aes/amiutil/crypto (LoadError)
from /usr/lib/site_ruby/aes/amiutil/uploadbundle.rb:1
```

See Also:

Amazon’s Developer Guide - Bundling an AMI: «If you receive a load error when running one of the AMI utilities, Ruby might not have found the path. To fix this, add /usr/lib/site_ruby to Ruby’s library path, which is set in the RUBYLIB environment variable.»

So, before using the *AMI Tools*, we have to add `/usr/lib/site_ruby` to `$RUBYLIB`. This should work:

```
$ export RUBYLIB=$RUBYLIB:/usr/lib/site_ruby
$ ec2-upload-bundle --help
```

We will set the environment variable `$RUBYLIB` automatically later on. The *AMI Tools* are now installed properly.

Note: The *API Tools* need *Java* installed. A version of at least 1.5 is required. We will use *Yum* to install it.

Install Java: Use *Yum* to install *Java*:

```
$ yum install java
```

Acknowledge to install `java-1.5.0-sun-compat` (in my case) and `jdk`.

Install and configure the API Tools: Download the *API Tools* as zip file and extract it:

```
$ cd /root
$ wget http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip
$ unzip ec2-api-tools.zip
```

The tools were extracted to a new directory e.g. `/root/ec2-api-tools-1.3-24159` (with the *API Tools* version in the name). There is no "installation" needed, but some further configuration.

Most of the commands provided by the tools need to know the path to "the user's PEM encoded RSA public key certificate file" and the path to "the user's PEM encoded RSA private key file". You can download these files from AWS' web interface after signing up successfully for *EC2*. The AMI you will bundle will be your own AMI and you will be the only one using instances of this AMI and you will almost for sure need these key files available in running instances more often. So it is no problem and even convenient to store these files in the virtual system and to bundle them into the AMI. `scp` them from anywhere (called `user@host:/path/to/.ec2`) to `/root/.ec2` on the *SL* virtual machine:

```
$ mkdir /root/.ec2
$ scp user@host:/path/to/.ec2/* /root/.ec2
    user@hosts's password:
    cert-**********************.pem     100%  916     0.9KB/s   00:00
    pk-**********************.pem       100%  926     0.9KB/s   00:00
```

Note: Why chosing `/root as directory for all the files`? *EC2* instances are created and terminated rapidly. If you make a mistake on an instance you can just terminate it and try again. For this reason working as root is not as dangerous as on a "normal" system. At first I worked with different users on *EC2* instances but I

realized that there was no need to. Finally the linux systems I used on *EC2* only knew one user: root. I store everything special that normally would be stored in a `home` directory in `/root`.

The *API Tools* need some environment variables to be set as explained in Amazons EC2 Getting Started Guide:

> **See Also:**
> Amazon's Getting Started Guide - Prerequisites: «The command line tools depend on an environment variable (JAVA_HOME) to locate the Java runtime. This environment variable should be set to the full path of the directory that contains a sub-directory named bin which in turn contains the java (on Linux/Unix) or the java.exe (on Windows) executable. You might want to simplify things by adding this directory to your path before other versions of Java.»
>
> **See Also:**
> Amazon's Getting Started Guide - Setting up the Tools: «The command line tools depend on an environment variable (EC2_HOME) to locate supporting libraries. You'll need to set this environment variable before you can use the tools. This should be set to the path of the directory into which the command line tools were unzipped. This directory is named ec2-api-tools-A.B-rrrr (A, B and r are version/release numbers), and contains sub-directories named bin and lib. [...] The environment variable EC2_PRIVATE_KEY should reference your private key file, and EC2_CERT should reference your X509 certificate.»

The best thing is to create a new file `/root/AWS_SET_ENV_VARS.sh` with the following content (customize it with your file- and directory names):

```
export JAVA_HOME=/usr
export EC2_HOME=/root/ec2-api-tools-*.*-*****
export EC2_PRIVATE_KEY=/root/.ec2/pk-**********************.pem
export EC2_CERT=/root/.ec2/cert-**********************.pem
export PATH=$PATH:$EC2_HOME/bin
```

Then just `source` the file and the needed variables are set:

```
$ source /root/AWS_SET_ENV_VARS.sh
```

Cleanup: Keep the system clean and slim because you have to pay for the amount of data you store on *S3*. Remove the archives and the folders they were extracted to that are not needed anymore (*tar*-archive and dir, *Ruby*-archive and dir, *API Tools* archive)

Note: In one of the early tests I bundled this system state into an AMI and tried to run it on *EC2*. But I could not connect to the running *EC2*- instance. Using the Amazon EC2 Query API-call `GetConsoleOutput` I could read the boot log. The graphical *Hardware*

Discovery Utility kudzu recognized a hardware change when booting my system on *EC2* but it did not reconfigure automatically. So `eth0` could not be brought up and for this reason there was no way to connect to the *EC2* instance. The two essential lines were:

```
Hardware configuration timed out.Run '/usr/sbin/kudzu' from the command line to re-detect.
```

and:

```
Bringing up interface eth0: Device eth0 has different MAC address than expected, ignoring.
[FAILED]
```

VMware emulates other hardware than *Xen*, which is the virtualization software running on *EC2*. So the change of MAC address makes sense. *kudzu* is able to reconfigure this, but it needs someone to press a key within 30 seconds. Nobody can interfere with a graphical boot application on a remote machine. This is adverse. After reading here and there and some testing I could solve this problem as you can read in the following part.

Modify kudzu and hardware configuration:

- Remove the complete `HWADDR`-line from `/etc/sysconfig/network-scripts/ifcfg-eth0`:

  ```
  $ cd /etc/sysconfig/network-scripts
  $ mv ifcfg-eth0 backup_ifcfg-eth0
  $ cat backup_ifcfg-eth0 | grep --invert-match HWADDR > ifcfg-eth0
  ```

- Remove the complete `class: NETWORK` from `/etc/sysconfig/hwconf` with e.g. *vi*. I deleted the following lines:

  ```
  -
  class: NETWORK
  bus: PCI
  detached: 0
  device: eth0
  driver: pcnet32
  desc: "Advanced Micro Devices [AMD] 79c970 [PCnet32 LANCE]"
  network.hwaddr: 00:0C:29:FA:B6:CB
  vendorId: 1022
  deviceId: 2000
  subVendorId: 1022
  subDeviceId: 2000
  pciType: 1
  pcidom:    0
  pcibus:  0
  pcidev: 10
  pcifn:  0
  ```

Note: What will happen on the next reboot (e.g. on *EC2*)? *kudzu* will recognize network hardware that is not listed in `/etc/sysconfig/hwconf`, the listing of current installed hardware. *kudzu* will try to configure it, but 30 seconds will pass without doing anything. After this, the ethernet hardware itself works very well, even without being configured by *kudzu*. The MAC address was deleted out of `ifcfg-eth0`, so the prior problem will disappear:

```
Bringing up interface eth0:  [  OK  ]
```

For this reason connecting to the remote machine using ssh will be possible. After logging in, manually invoking *kudzu* in the quiet mode will update the hardware listing without any (graphical) problems:

```
$ kudzu -q
```

After this, another reboot will be *kudzu*-free with a running device `eth0`. So this is the way we plan to do.

Warning: To ensure that the device `eth0` is brought up properly on *EC2* instance boot, **this state has to be bundled before the next reboot** is done (as explained in the note-box before).

7.3 Bundle, upload and register the AMI

In the following part I will describe how to **bundle** the *Scientific Linux* virtual machine. After bundling, we will **upload** the new *Amazon Machine Image* to Amazon's *Simple Storage S3* using the *AMI Tools*. To tell *EC2* that there is a new AMI stored in a *S3*-bucket, we will have to **register** the uploaded AMI using the *API Tools*.

7.3.1 Bundle

As discussed in *6.1.2 Two different ways to create an AMI* we will use `ec2-bundle-vol` to bundle the AMI. You should at first read the help:

```
$ export RUBYLIB=$RUBYLIB:/usr/lib/site_ruby
$ ec2-bundle-vol --help
```

Note: You do not want to set all environment variables like `RUBYLIB` manually anymore? Append `/root/AWS_SET_ENV_VARS.sh` and then `source` it again:

```
$ echo 'export RUBYLIB=$RUBYLIB:/usr/lib/site_ruby' >> /root/AWS_SET_ENV_VARS.sh
$ source /root/AWS_SET_ENV_VARS.sh
```

Build the command line options for ec2-bundle-vol:

I will show you the command line options you need step by step. The AMI will be encrypted using information from your *EC2* private key and certificate. So the first three options of every `ec2-bundle-vol` invoking are:

```
-k $EC2_PRIVATE_KEY     (path to private key file)
-c $EC2_CERT            (path to certificate file)
-u ************         (EC2 user id without hyphens)
```

Note: Set a `EC2_UID` environment variable. It will be usefull! Append `/root/AWS_SET_ENV_VARS.sh`:

```
$ echo 'export EC2_UID=************' >> /root/AWS_SET_ENV_VARS.sh
$ source /root/AWS_SET_ENV_VARS.sh
```

The bundle command needs to know the system architecture (one of i386, x86_64). Our VM for *ATLAS Computing* is a 32bit system. You have to submit the directory to save the image to and the name of the image. The image itself is will exist two times (once in a big image file and once in ~10 MB parts) in the image directory. I chose `/mnt` because it is excluded automatically from bundling. The name should be expressive because it will be the most important identifier later on.

```
-r i386                 (32bit arch)
-d /mnt                 (image directory)
-p SL47-AWSAC-base      (image prefix/name)
```

Now it is important to know that the following bundle will **not** happen on *EC2*. As stated in the help we have to deactivate inheriting instance metadata (in fact I do not know the benefit of inheriting). And: the bundle program must generate an `fstab` file. This is necessary to boot up properly on *EC2*.

```
--no-inherit            (do not inherit instance metadata)
--generate-fstab        (inject a generated EC2 fstab)
```

Invoke ec2-bundle-vol:

> **Warning:** Before invoking the command this is the last chance to check things and to change something before the system state is bundled.

At first check, if you really cleaned up installation files like archives or extracted archives.

Then let me tell you that setting some other environment variables is convenient:

Note: To upload files to *S3* - as it will happen when you invoke `ec2-upload-bundle` in the next step - you need the so-called *AWS Access Key ID* and the *AWS Secret Key*. Get them from your AWS webinterface and store them as environment variables to `/root/AWS_SET_ENV_VARS.sh`:

```
$ echo 'export AWS_ACCESS_KEY_ID=**********' >> /root/AWS_SET_ENV_VARS.sh
$ echo 'export AWS_SECRET_ACCESS_KEY=**********' >> /root/AWS_SET_ENV_VARS.sh
$ source /root/AWS_SET_ENV_VARS.sh
```

Now call `ec2-bundle-vol` with the commandline options derived above:

```
$ ec2-bundle-vol -k $EC2_PRIVATE_KEY -c $EC2_CERT -u $EC2_UID
                 --generate-fstab --no-inherit -r i386 -d /mnt -p SL47-AWSAC-base
```

Note: This special set of command line options is necessary only when you bundle an AMI at home. As soon as the image is stored on *S3* and rebundled from an *EC2* instance, the bundle command looks a bit different.

You will see an output like this:

```
Copying / into the image file /mnt/SL47-AWSAC-base...
Excluding:
     /var/lib/nfs/rpc_pipefs
     /sys
     /proc
     /proc/sys/fs/binfmt_misc
     /dev/pts
     /dev
     /media
     /mnt
     /proc
     /sys
     /mnt/SL47-AWSAC-base
     /mnt/img-mnt
1+0 records in
1+0 records out
mke2fs 1.35 (28-Feb-2004)
NOTE: rsync with preservation of extended file attributes failed.
Retrying rsyncwithout attempting to preserve extended file attributes...
/etc/fstab:
     # Legacy /etc/fstab
     # Supplied by: ec2-ami-tools-1.3-21885
     /dev/sda1 /     ext3    defaults 1 1
     /dev/sda2 /mnt  ext3    defaults 0 0
     /dev/sda3 swap  swap    defaults 0 0
     none      /proc proc    defaults 0 0
     none      /sys  sysfs   defaults 0 0
Bundling image file...
```

```
Splitting /mnt/SL47-AWSAC-base.tar.gz.enc...
Created SL47-AWSAC-base.part.00
...
Generating digests for each part...
Digests generated.
Creating bundle manifest...
ec2-bundle-vol complete.
```

When bundling is complete, you have an *Amazon Machine Image* of your *Scientific Linux* system in your local virtual file system. It consists of several *part files* like `SL47-AWSAC-base.part.00` and one *manifest file* like `SL47-AWSAC-base.manifest.xml`. In the next step these files will simply be uploaded to Amazon *Simple Storage S3* using a command line tool Amazon delivers.

7.3.2 Upload

Use `ec2-upload-bundle` to upload the *part files* and the *manifest file*. You should read the `-help` message at first. I will shortly explain the needed command line options. To interact with *Simple Storage S3*, the *AWS Access Key ID* and the *AWS Secret Key* are needed. Submit them with `-a` and `-s`. As described *in the S3 introduction*, files uploaded to *S3* are stored as objects in buckets. You have to submit the bucket you want to store the AMI in with `-b`. Since the space of bucket names is a space all AWS users share, you have to find a bucket name that does not exist until now. I got "ATLAS" ;). At least `ec2-upload-bundle` needs to know which AMI to upload: submit the path to the *manifest file* with `-m`.

My command looks like:

```
$ ec2-upload-bundle -a $AWS_ACCESS_KEY_ID -s $AWS_SECRET_ACCESS_KEY
                    -b ATLAS -m /mnt/SL47-AWSAC-base.manifest.xml
```

You get an error message?

```
Server.RequestTimeTooSkewed(403):
The difference between the request time and the current time is too large.
Bundle upload failed.
```

Every HTTP request (which is calling an *Amazon Web Services API command* like e.g. an upload to *S3*) contains a timestamp. If this *client GMT* differs from the *server GMT* by more than 15 minutes, the server declines the request with error 403. Why does that happen to us? Because *Scientific Linux*'s kernel base interrupt rate has been increased. Together with virtualization this results in time running much too slow.

See Also:

http://www.gossamer-threads.com/lists/linux/kernel/494604 - there you can learn much more details.

I tried setting the time once before starting the upload. But then the error reappeared in the middle of the upload, because the VM's clock really runs very slow (note: my upload was very fast!). I did not want to modify the system with things using *ntp*. I needed a quick (and dirty) solution, because uploading an AMI from "a VM at home" will happen only once or seldom. So I wrote `settimeloop.sh`:

```
#!/bin/sh
# settimeloop.sh: Get a timestring from a web server and set date using this
# string. Repeat this endlessly.
while true ;
do
    wget -m -nd http://gehrcke.de/awsac/permstuff/time.php
    date --set="`cat time.php`"
    sleep 10
done
```

using the following `time.php`:

```
<?php
echo gmstrftime("%a %b %d %H:%M:%S GMT %Y");
?>
```

Note: `time.php` will stay on my server - so you can use it! Since it delivers GMT, it works for all timezones. The `date -set` command should process the delivered timestring successfully on every english linux system.

Now create (download) `/root/settimeloop.sh`, make it executable and run it in a new shell. Then re-invoke `ec2-upload-bundle`. Step by step:

Note: You can run `settimeloop.sh` in background using `&`, too. Then you can skip the following new-shell-step, but you should redirect the output to a file or to `/dev/null`. Run it like this: `$ ./settimeloop.sh > looplog 2>&1 &`

- Find out the `inet addr` (IP address) of device `eth0` on your VM using `ifconfig`. Open a new shellconnection to your VM from the virtualizing host system using this IP address. For me this was **(typing in a shell of my virtualizing HOST system - not in a shell of the VM)**:

  ```
  $ ssh root@172.16.30.137
  ```

- Download `settimeloop.sh`. Make it executable and run it:

```
$ cd /root
$ wget http://gehrcke.de/awsac/permstuff/settimeloop.sh
$ chmod u+x /root/settimeloop.sh
$ ./settimeloop.sh
```

- re-invoke `ec2-upload-bundle` from the primary shell of your VM (the same command like before)

In my case the output looks like:

```
Uploading bundled image parts to https://s3.amazonaws.com:443/ATLAS ...
Uploaded SL47-AWSAC-base.part.00 to
   https://s3.amazonaws.com:443/ATLAS/SL47-AWSAC-base.part.00.
...
Uploading manifest ...
Uploaded manifest to https://s3.amazonaws.com:443/ATLAS/SL47-AWSAC-base.manifest.xml.
Bundle upload completed.
```

Depending on your internet connection this may take a very long time.

Note: Sometimes you may get other errors while uploading. If the connections breaks, you do not have to start from beginning.

While uploading various times, I got two different errors and the reason for them was not trackable for me:

```
Uploaded **** to https://s3.amazonaws.com:443/****
Error: failed to upload "****", Curl.Error(56):
SSL read: error:00000000:lib(0):func(0):reason(0), errno 104.
Bundle upload failed.
```

```
Uploaded ********* to https://s3.amazonaws.com:443/********
Error: failed to upload "***********", Server.InternalError(500)
: We encountered an internal error. Please try again.
Bundle upload failed.
```

In every case you can resume the download. Check out `ec2-upload-bundle -help`: you will see, that there is something like a "resume"-option: «`-part PART`: Upload the specified part and upload all subsequent parts.» This mostly worked for me instantly. But then you have to take care by yourself that every *part file* really is stored on *S3* successfully. You can recheck with the Firefox *S3* Extension S3Fox.

Lets say the last *part file* uploaded was `Uploaded *****.part.31`. Then just copy the upload command produced by `ami_upload` and append a `-part 32`:

```
ec2-upload-bundle -a $AWS_ACCESS_KEY_ID -s $AWS_SECRET_ACCESS_KEY -b ** -m ** --part 32
[...]
Skipping **.31.
Uploaded **.part.32 to https://s3.amazonaws.com:443/**.part.32.
[...]
Uploaded manifest to https://s3.amazonaws.com:443/**.manifest.xml.
Bundle upload completed.
```

When the last file, the *manifest file*, is uploaded, the new AMI on *S3* can be registered for use with *EC2*.

7.3.3 Register

To register the image you can use the *API Tools* command `ec2-register` or *Elasticfox*, that was first mentioned in *Manage and monitor EC2*. In both cases you need to know where your AMI *manifest file* is stored on *S3* (`bucketname/manifestfilename`). This is the command I invoked using the *API Tools*:

```
$ ec2-register -K $EC2_PRIVATE_KEY -C $EC2_CERT ATLAS/SL47-AWSAC-base.manifest.xml
```

EC2 checks, if the checksums listed in the *manifest file* correspond to the *part files*. On success *EC2* assigns a unique identification string (AMI ID) to the AMI you ordered to register. The command line tool returns this AMI ID:

```
IMAGE       ami-33d1355a
```

Now you have a registered AMI of your *Scientific Linux* system stored on *S3*. It is ready to start up in Amazon's *Elastic Computing Cloud*! You can shut down your local virtual machine using `shutdown -h now`.

7.4 Run, optimize and rebundle the AMI on EC2

In this part I will describe how to run an *EC2* instance of the new AMI. Then I will accomplish some unique optimization/modification of the AMI and finish in rebundling so that the modified system will be saved in a second AMI.

7.4.1 Run and connect

In a few moments you can feel as a manager of your own AMIs and *EC2* instances. For this you need a "management environment". I think you have seen enough of the *API Tools*. If you don't have any other possibilities, you can to use them. They are convenient if you e.g.

want to run an *EC2* command from within an instance. For this you installed and bundled them into the AMI. You now want to run an *EC2* instance of your new AMI. Of course, you can use the *API Tools* to run instances. But using them for every running, terminating, checking, registering, ... would soon get nerving.

I suppose that you have a graphical desktop environment (to read this documentation) and that you are able to use *Firefox* as browser. So I recommend to use *Elasticfox*, as I described in *Manage and monitor EC2*. Then you do not need to set up the *AMI Tools* on your local system, because *Elasticfox* supports all essential *EC2 API calls*. In the following I assume that you use *Elasticfox* (*S3Fox* for *S3 Simple Storage* is not bad, too) as "management environment".

In *Elasticfox*'s "Machine Images" list you will see all public AMIs and your AMIs. Use the filter field to filter out most of them (e.g. look for the bucket name you stored the new AMI in). Rightclick the new AMI and choose *Launch instance(s) of this AMI*. The default settings (e.g. security group `default` and instance type `m1.small`) are okay - so click *Launch*.

Refresh the "Your instances" list - the instance *State* should be *pending*. This means the AMI is processed and the virtual machine will start booting in some time (1-5 minutes). Use the time to set up *EC2 Network Security* to allow external requests (from "the internet") on port 22 to your instances. By default every port is blocked. Do you remember the security group setting before launching the instance? This was the `default` group. Change it in the *Security Groups* tab of *Elasticfox*: for group `default` grant permission for traffic from `CIDR 0.0.0.0/0` from port `22` to port `22` for `TCP/IP`. With this setting you will be able to connect to running instances in the `default` group using `ssh`.

When the instance *State* changes to *running* this means the virtual machine of your AMI has started booting. After waiting another 1-3 minutes, you can try to connect to the instance using the *public DNS* name. Rightclick the instance in the instances list and choose *Copy Public DNS Name to clipboard*. Then try connecting to your instance as root:

```
$ ssh root@ec2-75-101-217-23.compute-1.amazonaws.com
```

This should result in:

```
ssh root@ec2-75-101-217-23.compute-1.amazonaws.com
The authenticity of host [...]
Are you sure you want to continue connecting (yes/no)? yes
[...]
root@ec2-75-101-217-23.compute-1.amazonaws.com's password:
```

Enter your root password - congratulation, you are now logged in on an virtual machine of your own *Scientific Linux* system on *EC2*.

Note: If you can not connect to your instance for a long time, then use the *Show console output* option for your instance in *Elasticfox*. It allows you to see the output of an instance. The output does not get to you in realtime. There is some delay. *Show console output* is very useful to debug the instance boot (in fact it is the only way).

7.4.2 Optimize

Show console output lists some errors and warnings while booting the new AMI. In this paragraph I will describe how to remove the causes of these and how to make using the new AMI more comfortably. Some of the things I show you are required for some special functionality, some are for cleaning up the system and some are for comfort only. I recommend to carry out each step.

Reconfigure hardware: As described *here*, there is one last step missing to solve the *Kudzu*-problem completely. Execute *Kudzu* in quiet mode:

```
$ kudzu -q
```

Add kernel modules: After optimizing the running instance we want to bundle it into a new AMI using `ec2-bundle-vol`. But this now would result in:

```
Could not find any loop device.
Maybe this kernel does not know about the loop device? (If so, recompile or
   'modprobe loop'.)
```

After some web search I found a blogpost from Scott Parkerson solving the problem. Download the modules of the specific *EC2* kernel build 2.6.16-xenU from Amazon's Fedora Core 4 AMI:

```
$ wget http://people.rpath.com/~scott/cabinet/ec2/2.6.16-xenU-modules.tar.bz2
```

I mirrored the file: http://gehrcke.de/awsac/permstuff/2.6.16-xenU-modules.tar.bz2

Extract the archive to `/` (it then places all files to `/lib/modules/2.6.16-xenU`) and remove it:

```
$ tar xvjf 2.6.16-xenU-modules.tar.bz2 -C /
$ rm 2.6.16-xenU-modules.tar.bz2
```

Add the `loop` module:

```
$ modeprobe loop
```

Deactivate Thread-Local Storage: During the boot, the following warning appears:

```
** WARNING: Currently emulating unsupported memory accesses  **
**          in /lib/tls glibc libraries. The emulation is    **
**          slow. To ensure full performance you should      **
**          install a 'xen-friendly' (nosegneg) version of   **
**          the library, or disable tls support by executing **
**          the following as root:                           **
```

```
**          mv /lib/tls /lib/tls.disabled                    **
** Offending process: init (pid=1)                           **
```

So it makes sense to deactivate `tls`:

```
$ mv /lib/tls /lib/tls.disabled
```

Modify runlevel 4: As you can see in *console output*, *EC2* instances start up with runlevel 4. The current instance is overloaded with services I think you will not need. I deactivated the following:

```
$ cd /etc/rc.d/rc4.d
$ mv S05kudzu backup_S05kudzu
$ mv S80sendmail backup_S80sendmail
$ mv S09isdn backup_S09isdn
$ mv S09pcmcia backup_S09pcmcia
$ mv S40smartd backup_S40smartd
$ mv S18rpcidmapd backup_S18rpcidmapd
```

I could have deactivated more services. Feel free to extend this list (if you know what you are doing).

Autoset environment: In the previous paragraphs some various environment variables were needed. They will be needed in the future, too. So we will configure the system to automatically set the environment variables listed in `/root/AWS_SET_ENV_VARS.sh` at login. At first lets check whether your `/root/AWS_SET_ENV_VARS.sh` is complete. It should define the following variables (with partly different values):

```
export JAVA_HOME=/usr
export EC2_HOME=/root/ec2-api-tools-1.3-24159
export EC2_PRIVATE_KEY=/root/.ec2/pk-**********.pem
export EC2_CERT=/root/.ec2/cert-**********.pem
export PATH=$PATH:$EC2_HOME/bin
export RUBYLIB=$RUBYLIB:/usr/lib/site_ruby
export EC2_UID=******************
export AWS_ACCESS_KEY_ID=******************
export AWS_SECRET_ACCESS_KEY=******************
```

Then make this file sourced when logging in as root:

```
$ echo source AWS_SET_ENV_VARS.sh >> /root/.bashrc
```

I like `ll` to show all files in a special way, so this is the right moment to set this alias and/or other aliases:

```
$ echo "alias ll='ls -lah --color'" >> /root/.bashrc
```

You may now check if all the changes you made on the running instance until here work. `reboot` the instance:

```
$ reboot
```

Note: Invoking the standard `reboot` command keeps the current *EC2-instance* running. This is like restarting a computer. So all changes you made in the running instance will not get lost. The AMI is not loaded again. This would happen when you shut down/terminate the instance and launch a new one from the AMI. But keep in mind that, if there are any problems while rebooting, modified data will be lost.

After some time check the *console output* using *Elasticfox*. *Kudzu* should say nothing, the TLS-warning should be gone, some services should not boot up and the `modprobe: FATAL: Could not load /lib/modules/2.6.16-xenU/modules.dep`-error should be gone. Reconnect to the instance as root. By entering `env` you will see, that the special environment variables from `AWS_SET_ENV_VARS.sh` were set automatically.

Install useful scripts: When you often rebundle an image on *EC2*, then typing the whole `ec2-bundle-vol-` and `ec2-upload-bundle`-commands will get nerving. I will give you support for this with small *Python* scripts which are using the automatically set environment variables:

```
$ wget http://gehrcke.de/awsac/permstuff/AMIutils/root/ami_bundle
$ wget http://gehrcke.de/awsac/permstuff/AMIutils/root/ami_upload
$ wget http://gehrcke.de/awsac/permstuff/AMIutils/root/ami_delete
$ chmod u+x ami_bundle ami_delete ami_upload
```

I will describe the usage after the next step.

Cleanup: In the next step the instance state will be bundled into a new AMI. So this is the right moment to look for trash like archives that are no more needed and so on. Delete everything that is not needed any more.

7.4.3 Rebundle, upload, register

At first invoke the *Python* script `ami_bundle` I deliver for bundling:

```
$ ./ami_bundle
```

The usage should be self-explanatory. Enter an expressive AMI name. It should contain a version number because you almost for sure will change the AMI in the future. Bundle the image to `/mnt`. This folder is excluded from bundling. For me it looks like:

```
================ a script to invoke ec2-bundle-vol ================
     make sure that $EC2_PRIVATE_KEY, $EC2_CERT, $EC2_UID are set
===================================================================
```

```
image name: SL47-AWSAC-v01
image dir: /mnt

command:
ec2-bundle-vol -k $EC2_PRIVATE_KEY -c $EC2_CERT -u $EC2_UID
               -r i386 --no-inherit -d /mnt -p SL47-AWSAC-v01

execute? (y/n): y
Copying / into the image file /mnt/SL47-AWSAC-v01...
[...]
Created SL47-AWSAC-v01.part.56
Generating digests for each part...
Digests generated.
Creating bundle manifest...
ec2-bundle-vol complete.
```

Then invoke the *Python* script I deliver for uploading. The script needs to know the *S3*-bucket to store the AMI in. Submit this with `-b`. I recommend to take the same bucket like for the first AMI. Additionally the script needs the path to the *manifest*-file of the new AMI stored in the local file system of your current instance. Use TAB so that you do not have to type the whole path (for this reason I decided to use command line options for `ami_upload`). For me this looks like this:

```
$ ./ami_upload -b ATLAS -m /mnt/SL47-AWSAC-v01.manifest.xml


================== a script to invoke ec2-upload-bundle ======================
make sure that $EC2_HOME, $AWS_ACCESS_KEY_ID and $AWS_SECRET_ACCESS_KEY are set

          usage: -b S3bucketName -m PathToImageManifest
==============================================================================


command:
ec2-upload-bundle -a $AWS_ACCESS_KEY_ID -s $AWS_SECRET_ACCESS_KEY
                  -b ATLAS -m /mnt/SL47-AWSAC-v01.manifest.xml

execute? (y/n): y
Uploading bundled image parts to https://s3.amazonaws.com:443/ATLAS ...
[...]
Uploaded manifest to https://s3.amazonaws.com:443/ATLAS/SL47-AWSAC-v01.manifest.xml.
Bundle upload completed.
```

Note: If the connections breaks, you do not have to start from beginning. As described *above*, there is a way to resume an upload.

The modified AMI is now stored on *S3*. After registering you can use it. Using *Elasticfox*, registering is really easy: rightclick a free area in *Elasticfox*'s "Machine Images" list and click *Register a new AMI*. Enter `bucket/manifestPath`. In my case this is `ATLAS/SL47-AWSAC-v01.manifest.xml`. After confirming, the new AMI will appear in the list.

Now you can shut down the instance of the "old" AMI:

```
$ shutdown -h now
```

7.5 Modify the AMI for AWSACtools

In the next steps the AMI should get prepared for the job system the *AWSACtools* deliver. So run an instance of your latest AMI (the one bundled *in the paragraph before*) and log in. *AWSACtools* mostly consist of *Python* scripts. Some of them use features of a newer *Python* version than the one that comes with *SL47*. So we will install a second, newer *Python*. *AWSACtools* use the *Python* module *boto* to invoke AWS API calls, as described in *4.1.3 Using the API for own applications*. We will install subversion to get the latest version of *boto*. After this the server autorun components of *AWSACtools* will be copied to the instance. *AWSACtools* will be injected into the system startup using `rc.local`. Then the instance will be bundled into a new AMI.

Install newer Python: Changing the distribution delivered *Python* is not a good idea. So let us install *Python 2.5.2* as an alternative installation (no hard links and no manual) to `/opt/bin/python`:

```
$ wget http://www.python.org/ftp/python/2.5.2/Python-2.5.2.tar.bz2
$ tar xjf Python-2.5.2.tar.bz2
$ cd Python-2.5.2
$ ./configure --prefix=/opt
$ make
$ make altinstall
$ cd ..
$ rm -rf Python-2.5.2
$ rm Python-2.5.2.tar.bz2
$ ln -s /opt/bin/python2.5 /opt/bin/python
```

Test the new *Python*:

```
$ /opt/bin/python
```

should result in something like this:

```
Python 2.5.2 (r252:60911, Oct 13 2008, 14:33:49)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-10)] on linux2
```

Install subversion: I make it short:

```
$ yum install subversion
```

Install boto: *boto* should be installed as a site package for the new *Python*. Download the latest revision of *boto* to `/root/boto` and then create a symbolic link to this folder in the `site-packages` directory of *Python*:

```
$ svn checkout http://boto.googlecode.com/svn/trunk/ /root/boto
$ ln -s /root/boto/boto /opt/lib/python2.5/site-packages
```

Note: `/root/boto/boto` is because the linked directory must contain the `__init__.py` and *boto*'s SVN tree has another subdirectory `boto`.

Now you can test *boto*. Run `/opt/bin/python` and enter:

```
>>> import boto
>>> conn = boto.connect_ec2()
>>> images = conn.get_all_images()
>>> print images
```

This will print a list of objects describing all public and your AMIs.

Note: *boto* uses the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` TO authenticate with AWS.

Install AWSACtools autorun: There is not much of *AWSACtools* that has to be bundled into the AMI since *AWSACtools* are designed to be flexible. It is only the beginning of a chain of events that will happen when a job session is started. At first download a modified rc.local and overwrite the original one:

```
$ wget http://gehrcke.de/awsac/permstuff/AWSACtools/
   awsac-autorun_rev001_for_chap6/etc/rc.d/rc.local
$ mv rc.local /etc/rc.d/rc.local
$ chmod u+x /etc/rc.d/rc.local
```

The sense of this new `rc.local` and of the two files downloaded in the next step is explained in *how AWSACtools work¶*.

Two more files are needed:

```
$ mkdir /root/awsac
$ cd /root/awsac
$ wget http://gehrcke.de/awsac/permstuff/AWSACtools/
   awsac-autorun_rev001_for_chap6/root/awsac/awsac-autorun.sh
$ wget http://gehrcke.de/awsac/permstuff/AWSACtools/
   awsac-autorun_rev001_for_chap6/root/awsac/getsessionarchive
```

This was everything.

Rebundle, upload, register: Cleaned everything up? Now bundle the instance state into a new AMI:

```
$ cd /root
$ ./ami_bundle
```

I chose:

```
image name: SL47-AWSAC-v02
image dir: /mnt
```

Every modification of an instance that you want to bundle in a new AMI is followed by *6.4.3 Rebundle, upload, register*... So, do it! :)

CHAPTER

EIGHT

HOW TO CREATE AN ATLAS SOFTWARE RELEASE EBS SNAPSHOT

In this chapter I will lead you to an *Elastic Block Store (EBS) snapshot* of the *ATLAS Software Release* version you wish to use on *EC2*. This snapshot can be used by the *AWSACtools job system*, described in chapter *5 The job system: how AWSACtools work*.

After launching an instance, an EBS will be created, attached to the instance and mounted into the file system. The desired release of the ATLAS Software will be installed using Pacman. Then a standard `cmt` configuration will be performed that lets you easy initialize your jobs later on. After this, of course, the EBS content is saved into an EBS snapshot to *S3*.

8.1 Preparation

I recommend to use *Elasticfox* to execute the following steps.

Launch an instance: At first launch an instance of e.g. your latest *AWSAC* AMI. Any running *EC2* instance is okay for the following.

Create EBS Volume: Use *Elasticfox* to create a new EBS volume. **It must be in the same availability zone like the instance just launched**. The cost for the EBS volume and also for the EBS snapshot depend on the size of the volume/snapshot. So the size should be chosen carefully. I decided to make an EBS volume of `16 GB`. ATLAS Software takes about `8 GB`. The maximum overhead then is about `8 GB` - a waste of money? I am not sure about the consequences of too few free space while working with ATLAS Software (especially when working on a workspace that is **not** on the EBS). The Kit Validation warns if there is less than `7 GB` free space:

```
None of the following alternatives are satisfied:
[freeMegsMinimum 7000 free Megabytes at .] is not available.
```

```
[WARNING, less than the minimum of 7G free is required to install release,
   carry on anyway?]
hasn't been asked. Package will not be installed
```

This message appears *after* installing. I do not know what is more important - the EBS cost or the warning. But for this documentation I decided to follow the secure way.

Attach EBS Volume: Use *Elasticfox* to attach the new EBS volume to the instance just launched. Use e.g. `/dev/sdh` as device name.

Mount EBS and make a file system: Connect to the instance via ssh. Create the folder `/mnt/atlas`, format the new device with `Ext 3` and mount it:

```
$ mkdir /mnt/atlas
$ mkfs.ext3 /dev/sdh
    mke2fs 1.35 (28-Feb-2004)
    /dev/sdh is entire device, not just one partition!
    Proceed anyway? (y,n) y
    [...]
$ mount /dev/sdh /mnt/atlas
```

Lets check, if `/mnt/atlas` really corresponds to `/dev/sdh`, the EBS:

```
$ df
    Filesystem           1K-blocks      Used Available Use% Mounted on
    /dev/sda1             10321208   1845820   7951100  19% /
    /dev/sda2            350891748    199372 332868096   1% /mnt
    /dev/sdh              16513960     77888  15597212   1% /mnt/atlas
```

Install Pacman: Now we need *Pacman*, the installer for the ATLAS Software. Download, extract and configure it:

```
$ wget http://physics.bu.edu/pacman/sample_cache/tarballs/pacman-latest.tar.gz
$ tar xf pacman-latest.tar.gz
$ cd pacman-3.26/
$ source setup.sh
```

8.2 Install and configure ATLAS Software Release

Install:

In the next steps, the ATLAS Software Release version `14.2.24` will be installed to `/mnt/atlas/14.2.24`. So the "root" directory of the EBS volume will be `14.2.24`.

```
$ mkdir /mnt/atlas/14.2.24
$ cd /mnt/atlas/14.2.24
```

It is time to start *Pacman* and to decide whether you want to perform the Kit Validation (KV) after installation or not. If you want to:

```
$ pacman -allow trust-all-caches -get am-BU:14.2.24+KV
```

If you do not want to perform KV:

```
$ pacman -allow trust-all-caches -get am-BU:14.2.24
```

Note: I instructed *Pacman* to use **BU** (Boston University) as download source for the ATLAS Software release, because from *EC2*'s point of view this is much faster than the **CERN** mirror (Amazon's *EC2* data centres are located at the east coast).

After some time the result (with Kit Validation) should look like:

```
am-BU:Generic:http://atlas-computing.web.cern.ch/atlas-computing/links/
   monolith//mnt/atlas/14.2.24
About to execute: ./KitValidation/*/share/KitValidation [...]

############################################################
##        Atlas Distribution Kit Validation Suite         ##
##                 01-10-2008  v1.9.18-1                  ##
##                                                        ##
## Alessandro De Salvo <Alessandro.DeSalvo@roma1.infn.it> ##
############################################################
Testing AtlasProduction 14.2.24
athena executable                              [PASSED]
athena shared libs                             [PASSED]
Release shared libraries                       [PASSED]
Release Simple Checks                          [  OK  ]
Athena Hello World                             [  OK  ]
MooEvent compilation                           [  OK  ]
/mnt/atlas/14.2.24/KV-14.2.24/tmp
DB Release consistency check                   [  OK  ]

################################################
##   AtlasProduction 14.2.24 Validation [  OK  ]
################################################
```

Now the ATLAS Software Release is stored on the EBS and ready to use.

Note: During Kit Validation of `14.2.24` I got those warnings:

```
#CMT> Warning: template <src_dir> not expected in pattern install_scripts
   (from TDAQCPolicy)
#CMT> Warning: template <files> not expected in pattern install_scripts
   (from TDAQCPolicy)
```

They are not important and can be ignored:

See Also:

ATLAS Computing Workbook: «With Release 14.2.23, you might get [...] [these] warnings which [...] can be ignored»

Configure:

Before you can start jobs using the ATLAS Software Release, you have to configure the linux environment you are working on. This is done by sourcing a `setup.sh` that was automatically created by the *configuration management tool* `cmt`. This `setup.sh` must be sourced in any job shell script at the beginning. I will show you how to create this `setup.sh` using `cmt`.

The command to cause `cmt` to create the `setup.sh` is `cmt config`. It must be executed in a *configuration directory* that contains one special configuration file, the so-called `requirements` file. In this file you can specify your individual configuration that is needed by your jobs. `cmt config` parses the `requirements` file, processes the contained information and creates the corresponding `setup.sh`.

See Also:

More information about the coherences stated above can be found here: ATLAS Computing Workbook - Setting up your account:

So we at first create the *configuration directory* `/mnt/atlas/14.2.24/cmthome`, then the `requirements` file and then we invoke `cmt config`.

```
$ mkdir /mnt/atlas/14.2.24/cmthome
$ cd /mnt/atlas/14.2.24/cmthome
$ vi requirements
```

The following content should ensure a seamless offline/standalone functionality of the ATLAS Software Release:

```
set CMTSITE STANDALONE
set SITEROOT /mnt/atlas/14.2.24
macro ATLAS_DIST_AREA ${SITEROOT}
apply_tag opt
apply_tag setup
apply_tag noTest
set CMTCONFIG i686-slc4-gcc34-opt
use AtlasLogin AtlasLogin-* $(ATLAS_DIST_AREA)
```

Modify the content (e.g. the release version number) to your needs and save the file.

See Also:

The options above, their meanings and - of course - many more options are described here: The AtlasLogin environment setup package (especially here: The AtlasLogin environment setup package - The Home Requirements File and here: The AtlasLogin environment setup package - Available tags)

The next step is to use `cmt` to process the `requirements` file. But before `cmt` can be used, it must be initialized itself by sourcing the corresponding `setup.sh` at `/mnt/atlas/14.2.24/CMT/LATEST_VERSION/mgr/setup.sh`. For me this was

```
$ source /mnt/atlas/14.2.24/CMT/v1r20p20080222/mgr/setup.sh
```

Now `cmt config` can be invoked. This must happen in the *configuration directory* where we just placed the new `requirements` file:

```
$ echo $PWD
/mnt/atlas/14.2.24/cmthome
```

Thats okay; we are at the right place. Now invoke `cmt config`:

```
$ cmt config
```

It should result in

```
----------------------------------------
Configuring environment for standalone package.
CMT version v1r20p20080222.
System is Linux-i686
----------------------------------------
Creating setup scripts.
Creating cleanup scripts.
```

Now there are some more files in the current directory (the *configuration directory*), produced by `cmt`:

```
$ ls
cleanup.csh  cleanup.sh  Makefile  requirements  setup.csh  setup.sh
```

The `setup.sh` is the objective we were looking at. This is the file that must be sourced at the beginning of any job shell script. Lets try it now:

```
$ source /mnt/atlas/14.2.24/cmthome/setup.sh -tag=14.2.24
```

```
#CMT> Warning: template <src_dir> not expected in pattern install_scripts
   (from TDAQCPolicy)
#CMT> Warning: template <files> not expected in pattern install_scripts
   (from TDAQCPolicy)
```

Now I will show two ways to "check", if this initialization worked properly. At first, there now should be many executables in the `$PATH` beginning with `csc`. Change the current directory to any (e.g. `/mnt`) and enter `csc` and press TAB two times.

```
$ cd /mnt
$ csc + TAB + TAB
```

This should result in something like this:

```
csc_4d_segment_performance.exe      csc_fullchain_trf.py
csc_addTruthJetMet_trf.py           csc_genAtlfast08_trf.py
csc_atlasG4_trf.py                  csc_genAtlfast_trf.py
csc_atlfast_trf.py                  csc_genAtlfastTwoStep08_trf.py
csc_beamgasmix_trf.py               csc_genAtlfastTwoStep_trf.py
csc_beamhalo_trf.py                 csc_MergeHIST_trf.py
csc_BSrecoESD_trf.py                csc_mergeHIT_trf.py
csc_BSreco_trf.py                   csc_modgen_trf.py
csc_buildTAG_trf.py                 cscope
csc_cavernbkg_trf.py                cscope-indexer
csc_cluster_performance.exe         csc_physVal_Mon_trf.py
csc_cosmic_cluster.exe              csc_physVal_trf.py
csc_cosmics_sim_trf.py              csc_RDOtoBS_trf.py
csc_cosmics_trf.py                  csc_readasciigen_trf.py
csc_digi_reco_trf.py                csc_recoAOD_trf.py
csc_digi_trf.py                     csc_recoESD_trf.py
csc_evgen08new_trf.py               csc_reco_trf.py
csc_evgen08_trf.py                  csc_segment_performance.exe
csc_evgen900_trf.py                 csc_simseg_builder.exe
csc_evgen_input_trf.py              csc_simulID_recoFastCaloSim_trf.py
csc_evgen_trf.py                    csc_simul_reco_trf.py
csc_evgenTruthJetMet08_trf.py       csc_simul_trf.py
csc_evgenTruthJetMet_trf.py         csc_writeasciigen_trf.py
```

The second thing is to check, if there are some ATLAS extensions loaded into the `cmt` path:

```
$ cmt show path
```

This should result in something like this:

```
# Add path /mnt/atlas/14.2.24/AtlasOffline/14.2.24 from initialization
# Add path /mnt/atlas/14.2.24/AtlasAnalysis/14.2.24 from ProjectPath
# Add path /mnt/atlas/14.2.24/AtlasSimulation/14.2.24 from ProjectPath
# Add path /mnt/atlas/14.2.24/AtlasTrigger/14.2.24 from ProjectPath
# Add path /mnt/atlas/14.2.24/AtlasReconstruction/14.2.24 from ProjectPath
# Add path /mnt/atlas/14.2.24/dqm-common/dqm-common-00-05-00 from ProjectPath
# Add path /mnt/atlas/14.2.24/AtlasEvent/14.2.24 from ProjectPath
# Add path /mnt/atlas/14.2.24/AtlasConditions/14.2.24 from ProjectPath
# Add path /mnt/atlas/14.2.24/AtlasCore/14.2.24 from ProjectPath
# Add path /mnt/atlas/14.2.24/DetCommon/14.2.24 from ProjectPath
# Add path /mnt/atlas/14.2.24/GAUDI/v19r9-LCG54g from ProjectPath
# Add path /mnt/atlas/14.2.24/tdaq-common/tdaq-common-01-09-03 from ProjectPath
# Add path /mnt/atlas/14.2.24/LCGCMT/LCGCMT_54g from ProjectPath
```

This should validate the content of your new job *configuration directory* `/mnt/atlas/14.2.24/cmthome`. The EBS content is ready to get backed up into an EBS snapshot.

8.3 Build the snapshot

There is no need to modify the EBS anymore. Unmount the corresponding hard disk `/dev/sdh`:

```
$ umount /dev/sdh
```

Then use *Elasticfox* to detach the EBS from your instance.

Note: You should not detach it before the hard disk is unmounted!!

Then rightclick the EBS volume in *Elasticfox* and choose to create a snapshot. This will last some time. You can terminate the instance in the mean time, but you should not delete the EBS during snapshot creation! **Remember/note the snapshot ID of the new snapshot!** After snapshot creation has finished, delete the EBS volume.

Now you have a job prepared ATLAS Software Release stored on S3. Recreation of an EBS from this snapshot is an instantaneous process, since needed data will be loaded "to a new EBS volume" in the background.

CHAPTER

NINE

CONCLUSION AND OUTLOOK

Cloud Computing with virtual machines could be of big value for scientific applications, since it could reduce personnel and cost effort in computer centres. This is because virtualization facilitates computer centres to concentrate on their primary task: providing hardware and computing power. Disturbing software problems leading in crashing machines are displaced to the users, who - in case of virtualization - benefit from the *machine image concept* that allows to just reincarnate a system from an image file.

It was argued, that the optimal solution for scientific computing would be an own computing cloud for the price of own hardware in combination with an additional and easy accessible commercial cloud, that offers big flexibility itself (like *Amazon Web Services* do). *Easy accessible* means that a standard *Cloud Computing API* should be able to control both computing clouds. This would bring along the huge advantage of being able to switch between clouds in an impressively easy way, resulting in a very convenient solution to balance out peaks of desired computing power.

Motivated by these promising assumptions, I started building a proof-of-principle job system on top of the reliable technical infrastructure of *Amazon Web Services* (AWS), using the *Elastic Computing Cloud EC2* including *Elastic Block Stores* (EBS), the *Simple Storage Service S3* and the *SimpleDB*. It could be shown that it is possible to move serious scientific computing (*ATLAS Computing* with an *ATLAS Software Release* on *Scientific Linux 4*) to the cloud. Hence, *ATLAS Cloud Computing* is, in principle, a promising substitute for classical *ATLAS Computing* in the *LHC Computing Grid*. This also should apply to any other scientific computing application.

In the future, the next step is the implementation of the concrete plan to rebuild the job system by only using the services *EC2* (without EBS) and *S3*. This step smooths the way in direction of *cloud portability*, since these two services are the basic *Cloud Computing* services that should exist as basic versions in any computing cloud. This *reduced* job system will be available to any other AWS user by using public *Amazon Machine Images* and storing *ATLAS Software Releases* centrally on *S3* - which will increase usability.

And - to mention the best thing at the end - there already exists an approach to set up an *EC2* style cloud with own hardware: http://workspace.globus.org. The great *Nimbus CloudKit*, developed by the Nimbus people, makes it possible. Of course, *Nimbus* does not support the whole *EC2* API. This is the reason why the *AWSAC* job system must be reduced

to the elementary needed API calls, so that the systems will be able to match each other in the future.

Nimbus raises hope, that it is possible to develop a general *Cloud Computing API* with different possible clouds at the back end, e.g. a *Nimbus Cloud* for the price of own hardware and, in addition, Amazon's *EC2* to satisfy peaks of desired computing power.

Cloud Computing is coming and should come in science, too: as recently published (here - sorry, German only), the Steinbuch Centre for Computing - which provides the *LHC Tier 1 Centre* for Germany - cooperates with *Intel*, *HP* and *Yahoo* to explore the benefits of *Cloud Computing* for scientific applications, using a huge amount of hardware.

CHAPTER

TEN

APPENDIX

10.1 awsac-all-instances-autorun

```
#
#   ::::::::> awsac-all-instances-autorun (08-12-16) <::::::::
#
#   by Jan-Philip Gehrcke (jgehrcke@gmail.com)
#   Universität Würzburg, Max-Planck-Institut für Physik München
#
#   Copyright (C) 2008 Jan-Philip Gehrcke
#
#   LICENSE:
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 3 of the License, or
#   (at your option) any later version. This program is distributed in
#   the hope that it will be useful, but WITHOUT ANY WARRANTY; without
#   even the implied warranty of MERCHANTABILITY or FITNESS FOR A
#   PARTICULAR PURPOSE. See the GNU General Public License for more
#   details. You should have received a copy of the GNU General Public
#   License along with this program; if not, see
#   <http://www.gnu.org/licenses/>.
#
#   read the detailled documentation at http://gehrcke.de/awsac
#
#
# THIS FILE IS SOURCED AS ROOT ON ALL INSTANCES OF YOUR JOB SESSION
# (invoked by awsac-autorun)
#
# You should be able to use the following environment vars
# (set by awsac-autorun):
# export ATLASDir=/mnt/atlas
# export ATLASworkingDir=/mnt/atlasworkarea
# export AWSACworkingDir=/mnt/awsac
```

```
# export SessionInfoDir=${AWSACworkingDir}/sessioninfo
# export SessionInfoFile=${SessionInfoDir}/awsac.sessinfo
# export SessionArchiveFile=${AWSACworkingDir}/sessionarchive.tar.bz2
# export AWSACAutostartFile=${AWSACworkingDir}/awsac_all_instances_autorun.sh
#
#########################################################################

echo -e "\n******* awsac-all-instances-autorun *******\n"
echo -e "\n*******          v08-12-16          *******\n"
echo "# getting instance-id (from meta-data server)..."
InstanceIDFile=${SessionInfoDir}/InstanceID
wget http://169.254.169.254/latest/meta-data/instance-id -O ${InstanceIDFile}
if [ $? -eq 0 ]; then
    echo instance id: $(cat ${InstanceIDFile})
    echo "running /opt/bin/python ${AWSACworkingDir}/awsac-processjobs.py --sessioninfofile ${S

    /opt/bin/python ${AWSACworkingDir}/awsac-processjobs.py --sessioninfofile ${SessionInfoFil

else
    echo error while retrieving instance-id from meta-data server
    if [ -e ${InstanceIDFile} ]; then
        rm -rf ${InstanceIDFile}
    fi
fi
```

10.2 awsac-autorun

/etc/rc.d/rc.local :

```
#!/bin/sh
#
# This script will be executed *after* all the other init scripts.
# You can put your own initialization stuff in here if you don't
# want to do the full Sys V style init stuff.

touch /var/lock/subsys/local

# AWSAC's Individual Startup System
# this script MUST be invoked this way.
su -l root -c "sh /root/awsac/awsac-autorun.sh"
```

/root/awsac/awsac-autorun.sh :

```
#
#   ::::::::> awsac-autorun (08-10-13) <::::::::
#
#   by Jan-Philip Gehrcke (jgehrcke@gmail.com)
#   Universität Würzburg, Max-Planck-Institut für Physik München
#
#   Copyright (C) 2008 Jan-Philip Gehrcke
#
#   LICENSE:
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 3 of the License, or
#   (at your option) any later version. This program is distributed in
#   the hope that it will be useful, but WITHOUT ANY WARRANTY; without
#   even the implied warranty of MERCHANTABILITY or FITNESS FOR A
#   PARTICULAR PURPOSE. See the GNU General Public License for more
#   details. You should have received a copy of the GNU General Public
#   License along with this program; if not, see
#   <http://www.gnu.org/licenses/>.
#
#   read the detailled documentation at http://gehrcke.de/awsac
#
###############################################################################

echo -e "************************************************************"
echo -e "  AWSACtools: autorun (Individual Instance Startup System)"
echo -e "    /root/awsac/awsac-autorun.sh invoked from rc.local"
echo -e "                    by Jan-Philip Gehrcke"
echo -e "                        v08-10-13"
echo -e "************************************************************\n\n"

echo "# setting up environment variables..."
export ATLASDir=/mnt/atlas
echo ATLASDir: ${ATLASDir}
export ATLASworkingDir=/mnt/atlasworkarea
echo ATLASworkingDir: ${ATLASworkingDir}
export AWSACworkingDir=/mnt/awsac
echo AWSACworkingDir: ${AWSACworkingDir}
export SessionInfoDir=${AWSACworkingDir}/sessioninfo
echo SessionInfoDir: ${SessionInfoDir}
export SessionInfoFile=${SessionInfoDir}/awsac.sessinfo
echo SessionInfoFile: ${SessionInfoFile}
export SessionArchiveFile=${AWSACworkingDir}/sessionarchive.tar.bz2
echo SessionArchiveFile: ${SessionArchiveFile}
export AWSACAutostartFile=${AWSACworkingDir}/awsac_all_instances_autorun.sh
echo AWSACAutostartFile: ${AWSACAutostartFile}
```

```
echo "# creating directories for AWSAC and ATLAS-Software in /mnt..."
echo creating AWSACworkingDir
mkdir ${AWSACworkingDir}
echo creating SessionInfoDir
mkdir ${SessionInfoDir}
echo creating ATLASDir
mkdir ${ATLASDir}
echo creating ATLASworkingDir
mkdir ${ATLASworkingDir}
echo "# getting user-data (should contain the sessioninfostring)..."
wget http://169.254.169.254/latest/user-data -O ${SessionInfoFile}
if [ $? -eq 0 ]; then
    echo "# setting AWSenvironment variables..."
    source /root/AWS_SET_ENV_VARS.sh
    echo "# running getsessionarchive -i ${SessionInfoFile} -o ${SessionArchiveFile}"
    echo "# to get sessionarchive from S3 bucket..."
    /opt/bin/python /root/awsac/getsessionarchive -i ${SessionInfoFile} -o ${SessionArchiveFil
    if [ $? -eq 0 ]; then
        echo "# untaring sessionarchive..."
        tar xjvf ${SessionArchiveFile} -C ${AWSACworkingDir}
            if [ $? -eq 0 ]; then
                echo "# trying to execute AWSAC autostart shellscript..."
                if [ -e ${AWSACAutostartFile} ]; then
                    source ${AWSACAutostartFile}
                else
                    echo ${AWSACAutostartFile} does not exist
                fi
            else
                echo error while untaring ${SessionArchiveFile}
            fi
    else
        echo "error in /opt/bin/python /root/awsac/getsessionarchive -i ${SessionInfoFile} -o $
    fi
else
    echo "error while retrieving user-data: AWSACtools: autorun end"
    if [ -e ${AWSACworkingDir} ]; then
        echo delete ${AWSACworkingDir}
        rm -rf ${AWSACworkingDir}
    fi
fi
```

/root/awsac/getsessionarchive :

```
# -*- coding: UTF-8 -*-
#
#   ::::::::> getsessionarchive (08-10-13) <::::::::
```

```
#
#   by Jan-Philip Gehrcke (jgehrcke@gmail.com)
#   Universität Würzburg, Max-Planck-Institut für Physik München
#
#   Copyright (C) 2008 Jan-Philip Gehrcke
#
#   LICENSE:
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 3 of the License, or
#   (at your option) any later version. This program is distributed in
#   the hope that it will be useful, but WITHOUT ANY WARRANTY; without
#   even the implied warranty of MERCHANTABILITY or FITNESS FOR A
#   PARTICULAR PURPOSE. See the GNU General Public License for more
#   details. You should have received a copy of the GNU General Public
#   License along with this program; if not, see
#   <http://www.gnu.org/licenses/>.
#
#   read the detailled documentation at http://gehrcke.de/awsac
#
# simple script to save session archive object from S3 to a local
# file. S3 objects are described by bucket+key, so the script
# needs three infos: bucket, objectkey, outputfile.
# the -i argument defines a sessioninfofile that delivers these.
# it must contain the following : sessID;bucket;archivefilename
#
# the script then assumes that the archive lays at:
#           bucket/sessID/archivefilename
#
# the script will write the archive to outputfile defined by -o
#
# ATTENTION: at this time no bucket-names with capital
# letters are supported!!!! (forbidden by boto)
#
#############################################################################

import sys, boto

nArgs = len(sys.argv)
if (nArgs < 5):
    sys.exit("\nargumenterror: \nusage: getsessionarchive.py -i /path/to/sessioninfofile -o ou

for i in range(nArgs):
    if sys.argv[i] == "-i": infofile = sys.argv[i+1]
    if sys.argv[i] == "-o": outfile = sys.argv[i+1]

# parse the infofile for data..
```

```
try:
    data = open(infofile).readline()
    data = data.split(';')
    sessID = ''.join(data[0].split())
    bucketname = ''.join(data[1].split())
    archivename = ''.join(data[2].split())
except:
    sys.exit('error in sessioninfofile. expecting `sessID;bucket;archivefilename` in '+infofil

key = sessID+'/'+archivename
conn = boto.connect_s3()
bucket = conn.get_bucket(bucketname)
k = boto.s3.key.Key(bucket)
k.key = key
k.get_contents_to_filename(outfile)
print "getsessionarchive.py: downloaded "+key+" from bucket "+bucketname+" to "+outfile
```

10.3 awsac-processjobs

```
# -*- coding: UTF-8 -*-
#
#   ::::::::> awsac-processjobs (08-12-15) <::::::::
#
#   by Jan-Philip Gehrcke (jgehrcke@gmail.com)
#   Universität Würzburg, Max-Planck-Institut für Physik München
#
#   Copyright (C) 2008 Jan-Philip Gehrcke
#
#   LICENSE:
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 3 of the License, or
#   (at your option) any later version. This program is distributed in
#   the hope that it will be useful, but WITHOUT ANY WARRANTY; without
#   even the implied warranty of MERCHANTABILITY or FITNESS FOR A
#   PARTICULAR PURPOSE. See the GNU General Public License for more
#   details. You should have received a copy of the GNU General Public
#   License along with this program; if not, see
#   <http://www.gnu.org/licenses/>.
#
#   read the detailled documentation at http://gehrcke.de/awsac
#
#############################################################################
```

```
import sys
import os
import time
import subprocess
import optparse
import tarfile
import csv
import boto

VERSION = 'v2008-12-15'

# set the jobs' CWD (they create their own jobnumber-subfolder):
MAINWORKINGDIR = '/mnt/atlasworkarea'
# here we will look for the job shell scripts (AWSACworkingDir):
AWSACworkingDir = '/mnt/awsac'
# here the EBS volumes will be mounted
ATLASDIR = '/mnt/atlas'

def main():
    """
    lead through the script step by step
    """

    # split stdout+stderr to 'standard-stdout' and to logfile
    # (with the help of 'Tee'-class)
    logfilepath = 'processjobs.log'
    log_fd = open(logfilepath,'w+')
    sys.stdout = Tee(sys.stdout, log_fd)
    sys.stderr = sys.stdout

    # parse commandline arguments and check user given files
    options = parse_args()
    files = check_usergiven_files(options)

    # start output and get information about this instance and its jobs
    timestring = time.asctime(time.localtime())
    print '\n****** awsac-processjobs.py start '+timestring+' ******\n'
    print '# parsing the sessioninfofile '+files['infofilepath']+' ...'
    session_info_dict = parse_sessioninfofile(files['infofilepath'])

    print '  Session ID: '+session_info_dict['sessID']+' ; ',
    print 'CoresPerInstance: '+session_info_dict['CoresPerInstance']
    print '# getting information about this instance from EC2...'
    thisinstance = check_instance(options.instance_id)
    thislaunchindex = thisinstance.ami_launch_index
    thisinstance_id = thisinstance.id
    thiszone = thisinstance.placement
```

```
print ('  the ami-launch-index of this instance '
       '('+thisinstance_id+') is '+str(thislaunchindex))
print '  the Availability Zone of this instance is '+thiszone
print '# parsing the jobsconfigfile '+files['jobsfilepath']+' ...'
jobinfodicts_list = parse_jobconfigfile(jobsfile=files['jobsfilepath'],
                                        launchindex=int(thislaunchindex),
                                        cpi=int(session_info_dict['CoresPerInstance']))
print str(jobinfodicts_list)

# handle EBS volume for this instance
print '# prepare the creation of EBS volume(s) from '+\
      'the needed ATLAS Release snapshot(s)...'
ebs_volumes_list = init_EBSvols(jobinfodicts_list, thisinstance_id, thiszone)

ebsvolumes = EBSVolumes(ebs_volumes_list)

print '# create EBS volume(s)...'
ebsvolumes.create_vols_from_snaps()

print '# attach EBS volume(s)...'
ebsvolumes.attach_volumes()

print '# mount EBS volume(s)...'
ebsvolumes.mount_volumes()

print '# this is the list of successfully mounted EBS - snapshots:'
mounted_snap_ids_list = ebsvolumes.get_mounted_snap_ids()
print '  '+str(mounted_snap_ids_list)

print '# the following jobs now have their desired EBS running in system:'
jobinfodicts_list = filter_jobs_by_snap_id(jobinfodicts_list, mounted_snap_ids_list)
print str(jobinfodicts_list)

# build joblist: EVERY planned job for THIS instance gets
# its 'Job'-object and is appended to the joblist
print '# initialize running jobs...'
joblist = []
for jobinfodict in jobinfodicts_list:
    joblist.append(Job(jobinfodict))

# create an object 'jobs' of class 'Jobs'.
# Gets the joblist to contain all planned jobs. Gets session information.
jobs = Jobs(joblist=joblist,
            session_id=session_info_dict['sessID'],
            bucketname=session_info_dict['bucketname'],
            launchindex=thislaunchindex,
            instance_id=thisinstance_id)
```

```
    # run the jobs. this method will create a subprocess for each job
    jobs.runjobs()

    # check the status of the jobs' subprocesses. do things like stdout+err saving,
    # uploading result-archives etc.. look down for details.
    # loop ends when all subprocesses have finished.
    jobs.checkjobs_loop()

    # print some results..
    jobs.verbose()

    # clean up the system
    print '# unmount EBS volume(s)...'
    ebsvolumes.unmount_volumes()

    print '# detach EBS volume(s)...'
    ebsvolumes.force_all_detach()

    print '# delete EBS volume(s)...'
    ebsvolumes.delete_all()

    # stop splitting stdout+stderr and save logfile of this script
    print '# close processjobs-logfile...'
    print '  it will then be bundled and uploaded. then EC2 will be instructed to terminate thi
    upload_processjobs_log(session_id=session_info_dict['sessID'],
                           bucketname=session_info_dict['bucketname'],
                           launchindex=thislaunchindex,
                           logfilehandle=log_fd,
                           logfilepath=logfilepath)

    # shut the system down
    print '# instructing EC2 to terminate my instance..'
    terminate_this_instance(thisinstance_id)


class EBSVolumes(object):
    """
    Manages a list of instances of class `EBSVolume`.

    Provides methods that operate on all `EBSVolume`-instances.
    """

    def __init__(self, ebs_volumes_list):
        self.ebs_volumes_list = ebs_volumes_list

    def create_vols_from_snaps(self):
```

```
        for ebsvolume in self.ebs_volumes_list:
            ebsvolume.create()

    def attach_volumes(self):
        for ebsvolume in self.ebs_volumes_list:
            ebsvolume.attach()

    def mount_volumes(self):
        for ebsvolume in self.ebs_volumes_list:
            ebsvolume.mount()

    def unmount_volumes(self):
        for ebsvolume in self.ebs_volumes_list:
            ebsvolume.unmount()

    def get_mounted_snap_ids(self):
        mounted_snap_ids_list = []
        for ebsvolume in self.ebs_volumes_list:
            if ebsvolume.mountsuccess:
                mounted_snap_ids_list.append(ebsvolume.snap_id)
        return mounted_snap_ids_list

    def force_all_detach(self):
        for ebsvolume in self.ebs_volumes_list:
            ebsvolume.force_detach()

    def delete_all(self):
        for ebsvolume in self.ebs_volumes_list:
            ebsvolume.delete()


class EBSVolume(object):
    """
    One object of this class represents one physical EBS Volume.
    """

    def __init__(self, snap_id, device, thisinstance_id, thiszone):
        self.snap_id = snap_id
        self.thiszone = thiszone
        self.device = device
        self.botovolumeobject = None
        self.thisinstanceId = thisinstance_id
        self.createsuccess = False
        self.mountsuccess = False
        self.attachsuccess = False
        self.detachsuccess = False
```

```
    def exists_snap_id(self):
        """
        Check if there exists an EC2 EBS snapshot with the given ID (using boto)

        @return: True or False
        """
        conn = boto.connect_ec2()
        snapshots = conn.get_all_snapshots()
        for snapshot in snapshots:
            if snapshot.id == self.snap_id:
                return True
        return False

    def create(self):
        """
        Create an EC2 EBS volume of the given snapshot ID (using boto).
        Try it for some time.
        """
        if self.exists_snap_id():
            conn = boto.connect_ec2()
            self.botovolumeobject = conn.create_volume(size=20,
                                                       zone=self.thiszone,
                                                       snapshot=self.snap_id)
            print '  instructed EC2 to create EBS from snapshot '+self.snap_id
            if self.botovolumeobject is not None:
                volumestatus = self.botovolumeobject.status
                volumeid_list = [self.botovolumeobject.id]
                for counter in range(30):
                    print '  status: '+volumestatus
                    if volumestatus == 'available':
                        self.createsuccess = True
                        break
                    if counter == 29:
                        print '  we will no longer wait. check EBS manually...'
                    time.sleep(3)
                    conn = boto.connect_ec2()
                    volumes = conn.get_all_volumes(volume_ids=volumeid_list)
                    for volume in volumes:
                        volumestatus = volume.status
        else:
            print 'processjobs.py: snapshot '+self.snap_id+' does not exist'

    def attach(self):
        """
        Attach a created EC2 EBS volume to the given device (using boto).
        Try it for some time.
        """
```

```
        if self.createsuccess:
            conn = boto.connect_ec2()
            volumestatus = conn.attach_volume(self.botovolumeobject.id,
                                              self.thisinstanceId,
                                              self.device)
            print '  instructed EC2 to attach volume '+self.botovolumeobject.id+' to '+self.dev
            volumeid_list = [self.botovolumeobject.id]
            for counter in range(30):
                print '  status: '+volumestatus
                if volumestatus == 'attached':
                    self.attachsuccess = True
                    break
                if counter == 29:
                    print '  we will no longer wait. check EBS manually...'
                time.sleep(3)
                conn = boto.connect_ec2()
                volumes = conn.get_all_volumes(volume_ids=volumeid_list)
                for volume in volumes:
                    volumestatus = volume.status

    def force_detach(self):
        """
        Force detaching an attached EC2 EBS volume (using boto).
        Try it for some time.
        """
        if self.attachsuccess:
            conn = boto.connect_ec2()
            volumestatus = conn.detach_volume(volume_id=self.botovolumeobject.id,
                                              instance_id=self.thisinstanceId,
                                              force=True)
            print '  instructed EC2 to detach volume '+self.botovolumeobject.id
            volumeid_list = [self.botovolumeobject.id]
            for counter in range(30):
                print '  status: '+volumestatus
                if volumestatus == 'available':
                    self.detachsuccess = True
                    break
                if counter == 29:
                    print '  we will no longer wait. check EBS manually...'
                time.sleep(3)
                conn = boto.connect_ec2()
                volumes = conn.get_all_volumes(volume_ids=volumeid_list)
                for volume in volumes:
                    volumestatus = volume.status

    def delete(self):
        """
```

```
        Delete a detached EC2 EBS volume (using boto). Try it for some time.
        """
        if self.detachsuccess:
            conn = boto.connect_ec2()
            deletestatus = conn.delete_volume(volume_id=self.botovolumeobject.id)
            print '  instructed EC2 to delete volume '+self.botovolumeobject.id
            for counter in range(30):
                print '  status: '+str(deletestatus)
                if deletestatus == True:
                    self.deletesuccess = True
                    break
                if counter == 29:
                    print '  we will no longer wait. check EBS manually...'
                time.sleep(3)
                conn = boto.connect_ec2()
                deletestatus = conn.delete_volume(volume_id=self.botovolumeobject.id)

    def mount(self):
        """
        Mount an attached EC2 EBS volume to given device using subprocess module.
        """
        if self.attachsuccess:
            argumentlist = ['mount',self.device,ATLASDIR]
            # mount-subprocess. collect stderr and stdout in PIPE (file-like-object)
            print '  invoke mounting: subprocess.Popen() with args '+str(argumentlist)
            sp = subprocess.Popen(args=argumentlist,
                                  stdout=subprocess.PIPE,
                                  stderr=subprocess.STDOUT)
            returncode = sp.wait()
            output = sp.stdout.read()
            print '  mount subprocess ended. returncode: '+str(returncode)
            if output:
                print 'stdout+stderr: '+output
            if returncode == 0:
                self.mountsuccess = True


    def unmount(self):
        """
        Unmount a mounted EC2 EBS volume from given device using subprocess module.
        """
        if self.mountsuccess:
            argumentlist = ['umount',self.device]
            # umount-subprocess. collect stderr and stdout in PIPE (file-like-object)
            print '  invoke unmounting: subprocess.Popen() with args '+str(argumentlist)
            sp = subprocess.Popen(args=argumentlist,
                                  stdout=subprocess.PIPE,
```

```
                                stderr=subprocess.STDOUT)
            returncode = sp.wait()
            output = sp.stdout.read()
            print '  umount subprocess ended. returncode: '+str(returncode)
            if output:
                print 'stdout+stderr: '+output


class Job(object):
    """
    An instance of class 'Job' contains information about one single job.
    """
    def __init__(self, jobinfodict):
        self.shellscript = jobinfodict['shscript']
        self.jobnr = jobinfodict['jobnr']
        self.workingdir = os.path.join(MAINWORKINGDIR, self.jobnr)
        self.outfilepath = os.path.join(self.workingdir,'stdouterr_job_'+self.jobnr+'.log')
        self.starttime = None
        self.endtime = None
        self.returncode = None
        self.executiontime = None
        self.my_subprocess = None
        self.sdb_itemname = None


class Jobs(object):
    """
    Manages all jobs (instances of class 'Job') of this EC2-instance.

    Provides methods for running all jobs, collecting and uploading their data to S3,
    updating jobs' status in AWS SimpleDB.
    """
    def __init__(self, joblist, session_id, bucketname, launchindex, instance_id):
        self.joblist = joblist
        self.session_id = session_id
        self.bucketname = bucketname
        self.launchindex = launchindex
        self.instance_id = instance_id

    def sdb_create_jobitems(self):
        """
        Use boto to create a new jobitem for each job in the SimpleDB domain
        'session ID'. Initialize these items with some attributes.
        """
        sdb = boto.connect_sdb()
        domain = sdb.create_domain(domain_name=self.session_id)
        for job in self.joblist:
```

```
        itemname = 'job'+job.jobnr
        newitem = domain.new_item(itemname)
        newitem['jobnr'] = job.jobnr
        newitem['status'] = 'pending'
        newitem['instance_id'] = self.instance_id
        newitem['launchindex'] = self.launchindex
        newitem['shellscript'] = job.shellscript
        job.sdb_itemname = newitem.name

    def sdb_update_job(self, job, status):
        """
        Use boto to update the status of one single 'job' in SimpleDB:
        (re)set the attribute 'status' and add some more information to the DB.
        """
        sdb = boto.connect_sdb()
        domain = sdb.create_domain(domain_name=self.session_id)
        item = domain.get_item(job.sdb_itemname)
        if status == 'runstart':
            item['status'] = 'running'
            item['runstarttime'] = time.strftime('%y-%m-%d %H:%M:%S',time.localtime())
        if status == 'runend':
            item['status'] = 'waitforsave'
            item['runendtime'] = time.strftime('%y-%m-%d %H:%M',time.localtime())
            item['returncode'] = job.returncode
        if status == 'savestart':
            item['status'] = 'saving'
            item['savestarttime'] = time.strftime('%y-%m-%d %H:%M:%S',time.localtime())
        if status == 'saveend':
            item['status'] = 'finished'
            item['saveendtime'] = time.strftime('%y-%m-%d %H:%M:%S',time.localtime())


    def upload_results_archive(self, jobnr):
        """
        Look for results.tar.bz2 in the workingfolder ob job with 'jobnr'.
        Upload the file to S3 using boto to
        bucket/session_id/results_job_jobnr.tar.bz2
        """
        for job in self.joblist:
            if job.jobnr == jobnr:
                tarfilepath = os.path.join(job.workingdir,'results.tar.bz2')
                if os.path.exists(tarfilepath):
                    print '  found '+tarfilepath+' ('+str(os.path.getsize(tarfilepath))+' Byte
                    key = self.session_id+'/results_job_'+job.jobnr+'.tar.bz2'
                    S3_upload_file(file=tarfilepath,
                                   bucketname=self.bucketname,
                                   keyname=key)
```

```
                else:
                    print '  jobresultsarchive does not exist: '+tarfilepath

    # write stdout+stderr of job into logfile. compress logfile. upload archive.
    def create_upload_stdouterr_archive(self, jobnr):
        """
        Look for stdout+stderr-`outfilepath` of job with `jobnr`.
        Bundle it as bz2 and upload it to S3 using boto to
        bucket/session_id/stdouterr_job_jobnr.tar.bz2
        """
        for job in self.joblist:
            if job.jobnr == jobnr:
                if os.path.exists(job.outfilepath):
                    logarchived = False
                    try:
                        print '  outputfile of job '+job.jobnr+' found: '+job.outfilepath
                        tarfilename = 'stdouterr_job_'+job.jobnr+'.tar.bz2'
                        tarfilepath = os.path.join(job.workingdir,tarfilename)
                        tar = tarfile.open(tarfilepath, 'w:bz2')
                        tar.add(job.outfilepath,os.path.basename(job.outfilepath))
                        tar.close()
                        logarchived = True
                        print '  bundled '+job.outfilepath+' to '+tarfilepath
                    except:
                        print '  error in archiving log of job '+job.jobnr
                    if logarchived:
                        print '  planing to upload '+tarfilepath+' ('+str(os.path.getsize(tarfi
                        key = self.session_id+'/'+tarfilename
                        S3_upload_file(file=tarfilepath,
                                       bucketname=self.bucketname,
                                       keyname=key)
                else:
                    print '  logfile of job '+job.jobnr+' does not exist: '+job.outfilepath

    def runjobs(self):
        """
        Create one SimpleDB item for each job
        Iterate through all `job`s (basically shellscripts):
         - create individual directory for each job /mnt/atlasworkarea/jobnr
         - set this dir as cwd when calling `/bin/sh shellscript` with
           subprocess.Popen() method.
         - stdout and stderr of a subprocess are redirected into `job.outfilepath`
        Starting subprocesses this way is non-blocking.
        Save `job`s subprocess-objects in the `my_subprocess`-attributes
        Invoke SimpleDB update for running jobs
        """
        print '  creating SimpleDB jobitems...'
```

```
        self.sdb_create_jobitems()
        for job in self.joblist:
            print '# preparing job '+job.jobnr+'...'
            print '  cwd for job: '+job.workingdir,
            try:
                os.makedirs(job.workingdir)
                print " (created)"
            except OSError:
                print " (exists)"
            argumentlist = ['/bin/sh',job.shellscript]
            # now start the subprocess. collect stderr
            # and stdout in PIPE (file-like-object)
            print '  calling subprocess.Popen() with args: '+str(argumentlist)
            job.starttime = time.time()
            outfile = open(job.outfilepath,'w')
            job.my_subprocess = subprocess.Popen(args=argumentlist,
                                                 stdout=outfile,
                                                 stderr=subprocess.STDOUT,
                                                 cwd=job.workingdir)
            print '  runstart: updating sDB item '+job.sdb_itemname
            self.sdb_update_job(job, 'runstart')

    def checkjobs_loop(self):
        """
        Wait for all subprocesses to finish.
        If one finishes, upload its produced data, save returncode, update SimpleDB
        """
        print '# wait for subprocesses to finish...'
        while True:
            time.sleep(0.5)

            # are there running processes left? if not: break loop!
            occupied = False
            for job in self.joblist:
                if job.my_subprocess is not None:
                    occupied = True
                    break
            if not occupied:
                print '# all subprocesses ended'
                break

            # check for returnstate of subprocesses...
            for job in self.joblist:
                if job.my_subprocess is not None:
                    returncode = job.my_subprocess.poll()
                    if returncode is not None:
                        job.returncode = returncode
```

```
                        job.endtime = time.time()
                        print '# subprocess for job '+job.jobnr+' ended.',
                        print ' returncode: '+str(job.returncode)
                        print '  runend: updating sDB item '+job.sdb_itemname
                        self.sdb_update_job(job, 'runend')
                        print '  savestart: updating sDB item '+job.sdb_itemname
                        self.sdb_update_job(job, 'savestart')
                        self.upload_results_archive(job.jobnr)
                        self.create_upload_stdouterr_archive(job.jobnr)
                        print '  saveend: updating sDB item '+job.sdb_itemname
                        self.sdb_update_job(job, 'saveend')
                        # mark subprocess as finished
                        job.my_subprocess = None

    def verbose(self):
        """
        Print information about all `job`s
        """
        print '# summary:'
        for job in self.joblist:
            job.executiontime = time.strftime("%H:%M", time.gmtime(job.endtime-job.starttime))
            print ''
            outstring = 'Jobnumber: '+job.jobnr+'\n'+\
                        'starttime: '+time.strftime('%y-%m-%d %H:%M:%S',time.localtime(job.sta
                        'endtime: '+time.strftime('%y-%m-%d %H:%M',time.localtime(job.endtime)
                        'executiontime: '+job.executiontime+'\n'+\
                        'returncode: '+str(job.returncode)
            print outstring
            print '----------------------'


def terminate_this_instance(thisinstance_id):
    """
    Use boto to invoke 'TerminateInstances' API-call for this EC2-instance
    """
    conn = boto.connect_ec2()
    for counter in range(30):
        try:
            termresponse = conn.terminate_instances(instance_ids=[thisinstance_id])
            return
        except:
            print '  terminating resulted in an error-response. try again'
        time.sleep(10)
    print '  tried terminating often enough.. check it manually'


def upload_processjobs_log(session_id, bucketname, launchindex, logfilehandle, logfilepath):
```

```
    """
    Set sys.stdout and sys.stderr to originals, close logfilehandler, archive the
    logfile and upload it to S3 appending the launchindex to the filename.
    """
    sys.stdout = sys.__stdout__
    sys.stderr = sys.__stderr__
    logfilehandle.close()
    tarfilepath = 'processjobslog_LI_'+launchindex+'.tar.bz2'
    tar = tarfile.open(tarfilepath, 'w:bz2')
    tar.add(logfilepath,os.path.basename(logfilepath))
    tar.close()
    print '  bundled '+logfilepath+' to '+tarfilepath
    S3_upload_file(file=tarfilepath,
                   bucketname=bucketname,
                   keyname=session_id+'/'+os.path.basename(tarfilepath))


def S3_upload_file(file, bucketname, keyname):
    """
    Upload given `file`(name) to S3 bucket `bucketname` with the key `keyname`
    """
    try:
        print time.strftime('%y-%m-%d %H:%M:%S',time.localtime())+\
              ':  start upload. bucket:'+bucketname+'; key:'+keyname
        conn = boto.connect_s3()
        bucket = conn.create_bucket(bucketname)
        k = boto.s3.key.Key(bucket)
        k.key=keyname
        k.set_contents_from_filename(file)
        print time.strftime('%y-%m-%d %H:%M:%S',time.localtime())+':  finished.'
    except:
        print 'error while uploading '+file+' to bucket:'+bucketname+' ; key:'+keyname


class Tee(object):
    """
    Delivers a write()-method that writes to two filedescriptors.

    One should be standard-stdout and the other should describe a real file.
    If sys.stdout is replaced with an instance of this `Tee`-class and sys.stderr is
    set to sys.stdout, all stdout+stderr of the script is collected to console and
    to file at the same time.
    """
    def __init__(self, stdout, file):
        self.stdout = stdout
        self.file = file
```

```
    def write(self, data):
        self.stdout.write(data)
        try:
          self.file.write(data)
          self.file.flush()
        except:
          pass


def filter_jobs_by_snap_id(jobinfodicts_list, mounted_snap_ids_list):
    """
    Check if the snap_ids in the jobinfodicts are in the list of mounted snap_ids

    @return: A list of jobinfodicts that only contains jobs with mounted snap_ids
    """
    return [dict for dict in jobinfodicts_list if dict['snap_id'] in mounted_snap_ids_list]



def parse_jobconfigfile(jobsfile, launchindex, cpi):
    """
    Determine the jobs for this instance from jobsconfigfile (given by commandline).

    jobsconfigfile contains information for all EC2-instances in this session.
    The rows must have following format: snap_id;JobScriptname;Njobs
    Read in the jobsconfigfile and save data in jobsdatadicts_list.

    Grab the jobs for THIS EC2-instance from jobsdatadicts_list.
    The algorithm for determining these jobs from launchindex and cores per instance
    is explained below.

    @return: A list of dictionaries. The list has the following format:
          [{'shscript':'/path/shellskriptA.sh','snap_id':'snap-a232329','jobnr':'3'},
           {'shscript':'/path/shellskriptB.sh','snap_id':'snap-a232329','jobnr':'4'}]
    """
    jobsdatadicts_list = []
    try:
        filehandler = open(jobsfile)
        csvreader = csv.reader(filehandler, delimiter=';')
        for jobsdata in csvreader:
            jobsdatadict = {}
            jobsdatadict['snap_id']=jobsdata[0]
            jobsdatadict['shellscript']=jobsdata[1]
            jobsdatadict['Njobs']=jobsdata[2]
            jobsdatadicts_list.append(jobsdatadict)
        print "# jobsdatadicts_list read from "+jobsfile
    except:
```

```
        sys.exit("could not read in data from jobsfile "+jobsfile)

    # myjobinfodicts_list must be of following format:
    # [{'shscript':'/path/shellskriptA.sh','snap_id':'snap-a232329','jobnr':'3'},
    #  {'shscript':'/path/shellskriptB.sh','snap_id':'snap-a232329','jobnr':'4'}]

    myjobinfodicts_list = []
    # Determine my job numbers from launch index and Cores Per Instance
    my_first_jobnumber = launchindex*cpi+1
    my_last_jobnumber = launchindex*cpi+cpi
    my_jobnumbers = range(my_first_jobnumber, my_last_jobnumber+1)
    print "# my job numbers are: "+str(my_jobnumbers)

    # now filter my jobs out of jobs.cfg
    checkjobs_begin_number = 1
    for jobsdatadict in jobsdatadicts_list:
        checkjobs_end_number_plus_1 = checkjobs_begin_number +\
                                      int(jobsdatadict['Njobs'])
        jobnumbers_of_current_jobsdata = range(checkjobs_begin_number,
                                               checkjobs_end_number_plus_1)
        # is one my job numbers in the list of job numbers of current job?
        for myjobnumber in my_jobnumbers:
              if myjobnumber in jobnumbers_of_current_jobsdata:
                  # found a jobsdatadict containing a job for me->save jobsdata
                  myjobinfodict = {}
                  myjobinfodict['shscript'] = AWSACworkingDir+'/'+\
                                              jobsdatadict['shellscript']
                  myjobinfodict['snap_id'] = jobsdatadict['snap_id']
                  myjobinfodict['jobnr'] = str(myjobnumber)
                  myjobinfodicts_list.append(myjobinfodict)
        # in the first run of this loop the jobs 1-Njobs_first (e.g. 1-5) have
        # to be checked. this is the way I do:
        # - checkjobs_begin_number = 1
        # - checkjobs_end_number_plus_1=checkjobs_begin_number+Njobs_first = 6
        # range(1,6) returns (1,2,3,4,5) - this is the list we want!
        # lets declare the first 5 jobs as processed after the first loop run.
        # in the second run of the loop we want range(6,6+Njobs_second)
        # procuding the list (6,7,...,6+Njobs_second-1).
        # so checkjobs_begin_number must be increased by Njobs_first:
        checkjobs_begin_number += int(jobsdatadict['Njobs'])
    return myjobinfodicts_list


def parse_sessioninfofile(sessioninfofile):
    """
    Parse the sessioninfofile (given by commandline).
```

```
    The file contains information about the session this EC2-instance belongs to.
    The one and only row in this file must have the following format:
    sessID;bucket;sessionarchivefilename;CoresPerInstance

    Read in the data and save it in a dictionary session_info_dict.

    @return: dictionary session_info_dict containing session information.
    """
    session_info_dict = {}
    try:
        data = open(sessioninfofile).readline()
        data = data.split(';')
        session_info_dict['sessID'] = ''.join(data[0].split())
        session_info_dict['bucketname'] = ''.join(data[1].split())
        session_info_dict['archivename'] = ''.join(data[2].split())
        session_info_dict['CoresPerInstance'] = str(int(''.join(data[3].split())))
    except:
        sys.exit(('error in sessioninfofile. expecting `sessID;bucket;'
                  'sessionarchivefilename;CoresPerInstance` in '+sessioninfofile))
    return session_info_dict


def check_instance(instance_id):
    """
    Check if instance_id (given by commandline) belongs to a running EC2-instance
    using boto. If this EC2-instance exists, return a boto `Instance`-Class object.
    If not, it makes no sense to go on with the script.

    @return: boto `Instance`-Class object representing THIS EC2-instance
    """
    conn = boto.connect_ec2()
    reservations = conn.get_all_instances()
    for reservation in reservations:
        for instance in reservation.instances:
            if instance.id == instance_id:
                return instance
    sys.exit('InstanceID given by commandline was not found '+\
             'in currently running instances.')


def check_usergiven_files(options):
    """
    Check if the files given by commandline exist. `checkfile`-fkt will raise errors.

    @return: A dictionary containing the absolute paths of the files.
    """
    infofile = None
```

```
    jobsfile = None
    infofilepath = checkfile(options.infofile)
    jobsfilepath = checkfile(options.jobsfile)
    return {'infofilepath': infofilepath,
            'jobsfilepath': jobsfilepath}


def parse_args():
    """
    Parse commandlineoptions using the optparse module and check them for the
    logical consistence.

    @return: optparse `options`-object containing the commandlineoptions
    """
    # the following part configures the OptionParser...
    version = 'AWSACtools: processjobs'
    description = ("job system example")
    parser = optparse.OptionParser(version=version,description=description)
    parser.add_option('--sessioninfofile', dest='infofile',
                      help='job session info file (with userdata string)')
    parser.add_option('--jobsfile', dest='jobsfile',
                      help='jobs.cfg (containing jobinfo)')
    parser.add_option('--instanceID', dest='instance_id',
                      help='ID of THIS instance')

    # now read in the given arguments (from sys.argv by default)
    (options, args) = parser.parse_args()

    # now check the logical consistence...
    if (options.infofile is None) or \
       (options.jobsfile is None) or \
       (options.instance_id is None):
        parser.error('--sessioninfofile FILE and --jobsfile FILE and '+\
                     '--instanceID XXX are needed!')

    # everything okay until here? seems so.. return the given options.
    return options


def checkfile(file):
    """
    Check if a given file exists and really is a file (e.g. not a directory)
    In errorcase the script is stopped.

    @return: the absolute path of the file
    """
    if not os.path.exists(file):
```

```
        sys.exit(file+' does not exist')
    if not os.path.isfile(file):
        sys.exit(file+' must be a file')
    return os.path.abspath(file)



def init_EBSvols(jobinfodicts_list, thisinstance_id, thiszone):
    """
    Creates list of correctly initialized EBSVolume-classobjects.

    Detect the different snap-IDs for the instances jobs. Ideally its only one!
    Store them in different_snap_ids.
    Iterate through different_snap_ids:
    - choose a device path from a list of possible devices that does not exist in
      filesystem
    - delete this device path from the list of possible devices
    - Construct a new instance of 'EBSVolume' with this chosen device and other
      information (availabilityzone, instanceid, snapshotid)
    - append this new instance of 'EBSVolume' to the list ebs_volumes_list

    Verbose the ebs_volumes_list

    @return: ebs_volumes_list
    """
    # at first lets collect all different snapshot IDs
    different_snap_ids = []
    for jobinfodict in jobinfodicts_list:
        if jobinfodict['snap_id'] not in different_snap_ids:
            different_snap_ids.append(jobinfodict['snap_id'])
    print '  detected following different snap_ids: '+str(different_snap_ids)

    # this is the list of devices we can populate with EBS
    devices = ['/dev/sdh1','/dev/sdh2','/dev/sdh3',
               '/dev/sdh4','/dev/sdh5','/dev/sdh6',
               '/dev/sdh7','/dev/sdh8','/dev/sdh9']

    # now create one object of EBSVolume for each individual snap_id.
    # assign an individual device to each individual EBSVolume.
    # after this loop we have a list ebs_volumes_list that contains information
    # about individual EBS volumes to create from 'snap_id' and to mount to 'device'.
    ebs_volumes_list = []
    for snap_id in different_snap_ids:
        if devices:
            for possibledevice in devices:
                if not os.path.exists(possibledevice):
                    device = possibledevice
```

```
                    break
            devices.remove(device)
        else:
            sys.exit('processjobs.py: more than 9 EBS per instance are currently not supported
        ebs_volumes_list.append(EBSVolume(snap_id=snap_id, device=device,
                                          thisinstance_id=thisinstance_id,
                                          thiszone=thiszone))
    print '  planing to assign snap_id(s) to following device(s):'
    for ebsvolume in ebs_volumes_list:
        print '  '+ebsvolume.snap_id+'->'+ebsvolume.device
    return ebs_volumes_list


if __name__ == '__main__':
    main()
```

10.4 awsac-session

```
# -*- coding: UTF-8 -*-
#
#   ::::::::> awsac-session (08-10-13) <::::::::
#
#   by Jan-Philip Gehrcke (jgehrcke@gmail.com)
#   Universität Würzburg, Max-Planck-Institut für Physik München
#
#   Copyright (C) 2008 Jan-Philip Gehrcke
#
#   LICENSE:
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of the GNU General Public License as published by
#   the Free Software Foundation; either version 3 of the License, or
#   (at your option) any later version. This program is distributed in
#   the hope that it will be useful, but WITHOUT ANY WARRANTY; without
#   even the implied warranty of MERCHANTABILITY or FITNESS FOR A
#   PARTICULAR PURPOSE. See the GNU General Public License for more
#   details. You should have received a copy of the GNU General Public
#   License along with this program; if not, see
#   <http://www.gnu.org/licenses/>.
#
#   read the detailled documentation at http://gehrcke.de/awsac
#
#############################################################################

import sys
```

```
import boto
import hashlib
import time
import os
import random
import optparse
import ConfigParser

VERSION = "v08-10-13"

def main():
    """
    lead through the script step by step
    """

    print '\n'
    print '::::>  AWSACtools session management ' + VERSION
    print '::::>  by Jan-Philip Gehrcke'
    print '\n'

    options = parseargs()
    files = check_usergivenfiles(options)

    if options.action == 'start':
        print '# starting new session; parsing start information file... \n'
        s = Session(startinfo=parse_startinfofile(files['startinfofilepath']))
        print '# ----- shortdescription of new session: '+s.shortdescr.upper()\
              +' -----\n'

        s.calc_n_instances()
        print ('# instancetype chosen: `'+s.instance_type+'` with '+s.cores_per_instance+' '
               'core(s) per instance. With '+s.n_jobs+' demanded job(s)/core(s) '
               'this makes '+s.n_instances+' instance(s) we have to start. '
               'Unused cores: '+s.unused_cores+'.\n')

        print '# generating session ID from date, shortdescription, randomness:\n',
        s.gen_id()
        print s.session_id+'\n'

        print '# uploading sessionarchive file to S3...'
        if s.s3_upload_file_to_sessionfolder(files['archivefilepath']):
            print 'uploaded to S3 as object (bucket:`'+s.bucket+'` key:`'+s.session_id+'/'\
                                  +os.path.basename(files['archivefilepath'])+'`).\n'

        #s.s3_upload_datadir_to_sessionfolder(files['datadirpath'])

        print '# connecting to EC2 to check your AMI-ID...'
```

```
        s.check_image()
        print 'found AMI ``'+s.ami.id+'`` ('+s.ami.location+')\n'

        print '# building user-data string to be submitted to all instances...'
        s.build_runuserdata(files['archivefilepath'])
        print s.runuserdata+'\n(session_id;bucket;sessionarchivename;cores_per_instance)\n'

        print '# checking SimpleDB domain for this session...'
        s.sdb_delete_domain()

        print '\n# In the next step EC2 will be instructed to run exactly '\
               +s.n_instances+' instance(s) of given AMI (type ``'+s.instance_type\
               +'``) with user-data mentioned above.'
        if get_y_n("proceed? (y/n): ", ['y','n']) == "n":
            sys.exit('Then check your settings again (-: exiting.')
        if s.ec2_runinstances():
            print 'Request accepted. EC2 reservation ID: '+str(s.ec2_reservation.id)

        print '# saving session to file...',
        print s.save_to_file()

    if options.action == 'check':
        cfg = parse_configfile(files['configfilepath'])
        read_sessionstate_from_sdb(cfg['session_id'])

    if options.action == 'getresults':
        cfg = parse_configfile(files['configfilepath'])
        s3_download_files(bucketname=cfg['bucket'],
                          prefix=cfg['session_id'],
                          outputdir=os.path.join(files['outputdirpath'],'session-'+cfg['sessio

    if options.action == 'cleanup':
        cfg = parse_configfile(files['configfilepath'])
        deldomain = get_y_n('delete SimpleDB-domain '+cfg['session_id']+'? y/n: ', ['y','n'])
        if deldomain == 'y':
            sdb_delete_domain(cfg['session_id'])
        dels3keys = get_y_n('delete S3-objects '+cfg['bucket']+'/'+cfg['session_id']+'/* ? y/n
        if dels3keys == 'y':
            s3_delete_keys(bucketname=cfg['bucket'],
                           prefix=cfg['session_id'])


class Session:
    """
    Represents and manages one session.
    """
```

```
    def __init__(self, startinfo):
        self.session_id = None
        self.n_instances = None
        self.unused_cores = None
        self.cores_per_instance = None
        self.ami = None # will be a boto image object
        self.runuserdata = None
        self.ec2_reservation = None
        self.shortdescr=startinfo['shdescr']
        self.bucket=startinfo['bckt']
        self.n_jobs=startinfo['n_jobs']
        self.ami_id=startinfo['ami_id']
        self.ec2_uid=startinfo['ec2_uid']
        self.instance_type=startinfo['instance_type']


    def build_runuserdata(self, archivefile):
        """
        Build the user-data string and save it in attribute 'runuserdata'

        user-data must have following form:
        session_id;bucket;sessionarchivename;cores_per_instance
        """
        arcfilename = os.path.basename(archivefile)
        self.runuserdata = self.session_id+';'+self.bucket+';'+\
                           arcfilename+';'+self.cores_per_instance

    def calc_n_instances(self):
        """
        Calculate the number of EC2-instances in this session from n_jobs and n_cpi,
        the number of cores per instance.

        n_cpi is obtained from the instance-type.
        The number of unused cores is calculated, too (e.g. 3 jobs on c1.medium lead
        to one unused core).
        """
        if self.instance_type == 'm1.small':
            self.cores_per_instance = str(1)
        elif self.instance_type == 'c1.medium':
            self.cores_per_instance = str(2)
        n_jobs = int(self.n_jobs)
        cores_per_instance = int(self.cores_per_instance)
        n_instances = n_jobs / cores_per_instance
        overhead = n_jobs % cores_per_instance
        unused_cores = 0
        if overhead:
             n_instances += 1
```

```
            unused_cores = cores_per_instance - overhead
        self.n_instances = str(n_instances)
        self.unused_cores = str(unused_cores)

    def check_image(self):
        """
        Check if the usergiven amiID represents an existing Amazon Machine Image.

        If not, the script will quit.
        """
        ownerlist = []
        ownerlist.append(self.ec2_uid)
        myimage = None
        conn = boto.connect_ec2()
        myimages=conn.get_all_images(owners=ownerlist)
        for image in myimages:
            if image.id == self.ami_id:
                myimage = image
                break
        if myimage is None:
            sys.exit(("desired AMI `"+self.ami_id+"` was not in the list of"
                      " EC2-user's (UID: "+self.ec2_uid+") AMIs. exiting."))
        self.ami = myimage

    def ec2_runinstances(self):
        """
        Invoke the 'RunInstances" API-call using boto.

        @return: True. But: an error while API-call would result in exit.
        """
        conn = boto.connect_ec2()
        reservation=conn.run_instances(min_count=self.n_instances,
                                       max_count=self.n_instances,
                                       user_data=self.runuserdata,
                                       image_id=self.ami_id,
                                       instance_type=self.instance_type)
        self.ec2_reservation = reservation
        return True

    def save_to_file(self):
        """
        Save current session information to a file using ConfigParser.
        Put session id in filename.
        """
        filename = 'session-'+self.session_id+'.cfg'
        try:
            config = ConfigParser.SafeConfigParser()
```

```
            config.add_section('sessionconfig')
            config.set('sessionconfig', 'session_id', self.session_id)
            config.set('sessionconfig', 'ec2_reservation_id', str(self.ec2_reservation.id))
            config.set('sessionconfig', 'n_instances', str(self.n_instances))
            config.set('sessionconfig', 'unused_cores', str(self.unused_cores))
            config.set('sessionconfig', 'cores_per_instance', str(self.cores_per_instance))
            config.set('sessionconfig', 'runuserdata', self.runuserdata)
            config.set('sessionconfig', 'shortdescr', self.shortdescr)
            config.set('sessionconfig', 'bucket', self.bucket)
            config.set('sessionconfig', 'n_jobs', str(self.n_jobs))
            config.set('sessionconfig', 'ami_id', self.ami_id)
            config.set('sessionconfig', 'ec2_uid', str(self.ec2_uid))
            config.set('sessionconfig', 'instance_type', self.instance_type)
            fd = open(filename,'w')
            config.write(fd)
            fd.close()
        except:
            return 'error: could not write configfile: '+filename
        return filename

    def gen_id(self):
        """
        Generate session id from current time, shortdescription and randomhash
        """
        try: urandom = os.urandom(20)
        except: urandom = time.time()
        rnd = hashlib.sha1(str(random.random())+str(time.time())+str(urandom))
        sessionsuffix = rnd.hexdigest()[:4]
        timestring = time.strftime("%y%m%d_%H%M",time.localtime())
        self.session_id = timestring+"--"+self.shortdescr[:8]+"--"+sessionsuffix
        #self.session_id = "080910_1147--testsess--609f"

    def sdb_delete_domain(self):
        """
        Delete SimpleDB domain of current session
        """
        sdb_delete_domain(self.session_id)

    def s3_upload_file_to_sessionfolder(self, file):
        """
        Upload a file as S3 object to sessionsbucket/sessionid/file.

        @return: True (boto raises exceptions if something goes wrong)
        """
        conn = boto.connect_s3()
        bucket = conn.create_bucket(self.bucket)
        k = boto.s3.key.Key(bucket)
```

```
        k.key=self.session_id+"/"+os.path.basename(file)
        k.set_contents_from_filename(file)
        return True

    def s3_download_file_from_sessionfolder(self, getfile, outfile):
        """
        Download a file from S3 object sessionsbucket/sessionid/getfile
        to outfile.
        """
        conn = boto.connect_s3()
        bucket = conn.create_bucket(self.bucket)
        k = boto.s3.key.Key(bucket)
        k.key=self.session_id+"/"+getfile
        k.get_contents_to_filename(outfile)


def s3_delete_keys(bucketname, prefix):
    """
    Delete S3 objects beginning with `prefix` from `bucketname`.
    """
    conn = boto.connect_s3()
    bucketfound = False
    try:
        buckets = conn.get_all_buckets()
    except:
        print '  error while retrieving all buckets'
    if buckets:
        for bucket in buckets:
            if bucket.name == bucketname:
                bucketfound = True
                break
    if not bucketfound:
        print '  bucket could not be found: '+bucketname
    else:
        objectkeys = bucket.list(prefix=prefix)
        for key in objectkeys:
            key.delete()
            print 'deleted '+os.path.basename(key.name)


def sdb_delete_domain(domainname):
    """
    Delete SimpleDB domain `domainname`.
    """
    sdb = boto.connect_sdb()
    domains = ''
    try:
```

```
        domains = sdb.get_all_domains()
    except:
        print '  error while retrieving all domains'
    if domains:
        for domain in domains:
            if domain.name == domainname:
                sdb.delete_domain(domain)
                print '  domain existed and was deleted: '+domainname
                return
    print '  domain does not exist: '+domainname


def read_sessionstate_from_sdb(session_id):
    """
    Read SimpleDB domain `session_id` content and print it.

    Some special keynames are hardcoded in this function.
    """
    sdb = boto.connect_sdb()
    domainfound = False
    try:
        domains = sdb.get_all_domains()
    except:
        print '  error while retrieving all SimpleDB-domains'
    if domains:
        for domain in domains:
            if domain.name == session_id:
                domainfound = True
                break
    if not domainfound:
        print '  domain for this session could not be found: '+session_id
    else:
        # jobdict_list will be a list of jobdicts
        jobdict_list = []
        for jobitem in domain:
            # store all the key/value-pairs of the db-item in `jobdict`
            jobdict={}
            for key in jobitem.keys():
                jobdict[key] = jobitem[key]
            jobdict_list.append(jobdict)
        if jobdict_list:
            # lets sort the dictslist by a specific key in the dicts
            # I assume that the following keys exist but of course I will check it:
            # jobnr, status, instance_id, launchindex, shellscript, runstarttime,
            # runendtime, returncode, savestarttime, saveendtime
            jobdict_list.sort(key= lambda x: x['jobnr'])
            for jobdict in jobdict_list:
```

```
                try:
                    print '============= Job '+jobdict['jobnr']+' (status: '+jobdict['status']+
                except KeyError:
                    pass
                try:
                    print 'running '+jobdict['shellscript']+' on instance '+\
                          jobdict['instance_id']+' with launchindex '+jobdict['launchindex']
                except KeyError:
                    pass
                try:
                    print 'started running: '+jobdict['runstarttime']
                except KeyError:
                    pass
                try:
                    print 'ended running: '+jobdict['runendtime']
                except KeyError:
                    pass
                try:
                    print 'returncode: '+jobdict['returncode']
                except KeyError:
                    pass
                try:
                    print 'started saving: '+jobdict['savestarttime']
                except KeyError:
                    pass
                try:
                    print 'ended saving: '+jobdict['saveendtime']
                except KeyError:
                    pass
                print ''


def get_y_n(msg, commandlist):
    """
    Get user input from stdin using raw_input() with `msg` as prompt.

    Wait for any string that is in stringlist `commandlist`.
    @return: entered string (that is in `commandlist`)
    """
    while True: # wait for instring to be a known command
        while True: # wait for an errorfree input
            try:
                instring = raw_input(msg)
                break
            except:
                pass
        if instring in commandlist:
```

```
            return instring


def s3_download_files(bucketname, prefix, outputdir):
    """
    Download files from S3 objects `bucketname`/`prefix`* to `outputdir`.

    At first receive all buckets. Then get all objectkeys in `bucketname` with
    specified `prefix`. Download objects to `outputdir`. Ask for overwriting.
    """
    conn = boto.connect_s3()
    bucketfound = False
    try:
        buckets = conn.get_all_buckets()
    except:
        print '  error while retrieving all buckets'
    if buckets:
        for bucket in buckets:
            if bucket.name == bucketname:
                bucketfound = True
                break
    if not bucketfound:
        print '  bucket could not be found: '+bucketname
    else:
        try:
            outputdir = os.path.abspath(outputdir)
            os.makedirs(outputdir)
            print "outputfolder created: "+outputdir
        except OSError:
            print "outputfolder exists: "+outputdir
        overwriteall = False
        objectkeys = bucket.list(prefix=prefix)
        for key in objectkeys:
            outfilepath = os.path.join(outputdir,os.path.basename(key.name))
            if os.path.exists(outfilepath) and not overwriteall:
                instring = get_y_n(os.path.basename(key.name)+' exists. overwrite? y/n/a: ',['y
                if instring == 'n':
                    continue
                if instring == 'a':
                    overwriteall = True
            key.get_contents_to_filename(outfilepath)
            print 'saved '+os.path.basename(key.name)


def parse_configfile(file):
    """
    Parse given `file`name for session information/configuration.
```

```
    @return: return dict with special options if filecontent is as expected.
    """
    config = ConfigParser.SafeConfigParser()
    try:
        config.readfp(open(file))
        sesscfg = {}
        sesscfg['session_id'] = config.get('sessionconfig', 'session_id')
        sesscfg['ec2_reservation_id'] = config.get('sessionconfig', 'ec2_reservation_id')
        sesscfg['n_instances'] = config.get('sessionconfig', 'n_instances')
        sesscfg['unused_cores'] = config.get('sessionconfig', 'unused_cores')
        sesscfg['cores_per_instance'] = config.get('sessionconfig', 'cores_per_instance')
        sesscfg['runuserdata'] = config.get('sessionconfig', 'runuserdata')
        sesscfg['shortdescr'] = config.get('sessionconfig', 'shortdescr')
        sesscfg['bucket'] = config.get('sessionconfig', 'bucket')
        sesscfg['n_jobs'] = config.get('sessionconfig', 'n_jobs')
        sesscfg['ami_id'] = config.get('sessionconfig', 'ami_id')
        sesscfg['ec2_uid'] = config.get('sessionconfig', 'ec2_uid')
        sesscfg['instance_type'] = config.get('sessionconfig', 'instance_type')
        config.write(sys.stdout)
    except:
        sys.exit(('error while parsing '+file+' for configuration options.\n'
                  'expecting section [startinfo] with following options:\n'
                  'session_id,ec2_reservation_id,n_instances,unused_cores,cores_per_instance,\
                  'runuserdata,shortdescr,bucket,n_jobs,ami_id,ec2_uid,instance_type'))
    return sesscfg



def parse_startinfofile(file):
    """
    Check configuration file for options.

    Write the parsed data to stdout.
    @return: information as strings without whitespaces
    """
    config = ConfigParser.SafeConfigParser()
    try:
        config.readfp(open(file))
        sessinfo = {}
        sessinfo['shdescr'] = ''.join(config.get('startinfo','shortdescr').split())
        sessinfo['n_jobs'] = ''.join(config.get('startinfo','n_jobs').split())
        sessinfo['bckt'] = ''.join(config.get('startinfo','sessionsbucket').split())
        sessinfo['ami_id'] = ''.join(config.get('startinfo','ami_id').split())
        sessinfo['ec2_uid'] = ''.join(config.get('startinfo','ec2_uid').split())
        sessinfo['instance_type'] = ''.join(config.get('startinfo','instance_type').split())
        config.write(sys.stdout)
```

```
    except:
        sys.exit(('error while parsing '+file+' for configuration options.\n'
                  'expecting section [startinfo] with following options:\n'
                  'shortdescr,n_jobs,sessionsbucket,ami_id,ec2_uid,instance_type'))
    return sessinfo


# checks if the script really can work with user-given options.
# logical consistence has already been checked before, so lets look if
# the given files and directories exist the way we need it.
# returns dictionary of files and directories with absolute pathes.
def check_usergivenfiles(options):
    """
    Check if the files given by commandline exist. `check_file`-fkt will raise errors.

    @return: A dictionary containing the absolute paths of the files.
    """
    archivefilepath = None
    configfilepath = None
    startinfofilepath = None
    outputdirpath = None
    datadirpath = None
    if options.action == 'start':
        archivefilepath = check_file(options.archivefile)
        if not os.path.basename(archivefilepath).endswith("tar.bz2"):
            sys.exit(("The session archive file has to be a bz2 compressed tarball. The file "
                      "you submitted does not end with 'tar.bz2'. So I suppose wrong content."))
        startinfofilepath = check_file(options.startinfofile)
        if options.datadir is not None:
            datadirpath = check_dir(options.datadir)
    if options.action == 'check':
        configfilepath = check_file(options.configfile)
    if options.action == 'getresults':
        outputdirpath = check_dir(options.outputdir)
        configfilepath = check_file(options.configfile)
    if options.action == 'cleanup':
        configfilepath = check_file(options.configfile)
    return {'archivefilepath': archivefilepath,
            'configfilepath': configfilepath,
            'startinfofilepath': startinfofilepath,
            'outputdirpath': outputdirpath,
            'datadirpath': datadirpath}


def parseargs():
    """
    Parse commandlineoptions using the optparse module and check them for the
```

```
    logical consistence. Generate help- and usage-output.

    @return: optparse 'options'-object containing the commandlineoptions
    """
    # the following part configures the OptionParser...
    version = 'AWSACtools session management' + VERSION
    description = ("AWSAC session management" + VERSION + " - start sessions, check status "
                   "of sessions, get result data and clean up used AWS")
    usage = ("%prog [--start || --check || --getresults || --cleanup]\n\t\t"
             "[-a A -i I (-d D) || -c C || -c C -o O || -c C]\n\t\t"
             "type -h, --help for help and --version for versionoutput")
    dhelp = ('directory to grab additional session data from. this data will be made'
             ' available to the instances by uploading it to S3.'
             ' (-d is optional for 'start') WARNING: CURRENTLY NOT SUPPORTED')
    parser = optparse.OptionParser(usage=usage,version=version,description=description)
    parser.add_option('-a', '--archive', dest='archivefile',
                      help='session archive file (needed for session 'start')')
    parser.add_option('-i', '--ini', dest='startinfofile',
                      help='session startinformation file (needed for session 'start')')
    parser.add_option('-d', '--data', dest='datadir',
                      help=dhelp)
    parser.add_option('-c', '--config', dest='configfile',
                      help='session config file (needed for 'check', 'getresults', 'cleanup')'
    parser.add_option('-o', '--outdir', dest='outputdir',
                      help='directory to save results in (needed for 'getresults'; dir must ex
    parser.add_option('--start', action='store_const', const='start', dest='action',
                      help='start new session (needs ARCHIVEFILE and STARTINFOFILE set - may ha
    parser.add_option('--check', action='store_const', const='check', dest='action',
                      help='check status of an existing session (needs CONFIGFILE set)')
    parser.add_option('--getresults', action='store_const', const='getresults', dest='action',
                      help=('get result data of an existing session '
                            '(needs CONFIGFILE and OUTPUTDIR set)'))
    parser.add_option('--cleanup', action='store_const', const='cleanup', dest='action',
                      help='deletes all s3 objects of an existing session (needs CONFIGFILE se

    # now read in the given arguments (from sys.argv by default)
    (options, args) = parser.parse_args()

    # now check the logical consistence...
    if options.action is None:
        parser.error('one of [--start || --check || --getresults || --cleanup] must be set!')
    elif options.action == 'start':
        if (options.archivefile is None) or (options.startinfofile is None):
            parser.error('when --start is set, -a ARCHIVEFILE and -i STARTINFOFILE are needed!
    elif options.action == 'check':
        if (options.configfile is None):
            parser.error('when --check is set, -c CONFIGFILE is needed!')
```

```
    elif options.action == 'getresults':
        if (options.configfile is None) or (options.outputdir is None):
            parser.error('when --getresults is set, -c CONFIGFILE and -o OUTPUTDIR are needed!
    elif options.action == 'cleanup':
        if (options.configfile is None):
            parser.error('when --cleanup is set, -c CONFIGFILE is needed!')

    # everything okay until here? seems so.. return the given options.
    return options


def check_file(file):
    """
    Check if a given file exists and really is a file (e.g. not a directory)
    In errorcase the script is stopped.

    @return: the absolute path of the file
    """
    if not os.path.exists(file):
        sys.exit(file+' does not exist')
    if not os.path.isfile(file):
        sys.exit(file+' must be a file')
    return os.path.abspath(file)


def check_dir(dir):
    """
    Check if a given dir exists and really is a dir (e.g. not a file)
    In errorcase the script is stopped.

    @return: the absolute path of the directory
    """
    if not os.path.exists(dir):
        sys.exit(dir+' does not exist')
    if not os.path.isdir(dir):
        sys.exit(dir+' is not a directory')
    return os.path.abspath(dir)


if __name__ == "__main__":
    main()
```