Development of the AMIGA Front End for External Use

Toni Sagristà Sellés

Masters degree in Astrophysics, Particle Physics and Cosmology
Departament d'Astronomia i Meteorologia
Universitat de Barcelona

12 July 2011

Abstract

The AMIGA cosmological model offers an analytical solution to track the coupled evolution of galaxies and IGM since the inception of the Universe, but despite being so powerful it lacks a comprehensive and user-friendly way to access its output data and draw results easily. The immediate objective of this research project is to place a software layer on top of AMIGA in order to deal with the interactions between the final user and AMIGA itself. Thus, we have analyzed the issue and provided a solution in the form of a software front end, called AMIGA High-z Observatory, consisting of an interactive, usable, concurrent and efficient web application, which gathers users' requests and offers graphical and numerical responses based on AMIGA data. This, when publicly available on-line, shall represent the first of its kind in offering access to the data produced by a self-contained, fully-analytical cosmological model such as AMIGA.

Contents

List of Figures

Chapter 1

Introduction

The accelerated, flat, cold dark matter model (ΛCDM) provides a theoretical framework from which the problem of galaxy formation and evolution can be successfully tackled. In this framework, structure is thought to form and evolve from small-scale primordial density fluctuations to large-scale dark matter haloes through a series of successive mergers and continuous accretion in a well defined bottom-up hierarchical scenario [WR78] driven only by gravitational interactions. The result of this process produces a map of overdense and subdense regions resulting in the formation of potential wells that will soon start to capture the baryonic component, which is shock-heated in the fall into dark matter haloes. Once established, this baryonic gas irradiates, cools and condenses into baryonic structures such as stars and galaxies.

However, the details of this process are rather unclear and to a certain extent the process of galaxy formation is one of the most challenging unresolved issues in modern cosmology, altogether with the nature of dark matter and dark energy.

In order to shed some light on the topic, two main approaches have been developed, namely, *semi-analytic models* and *n-body/hydrodynamic simulations*. The former techniques start from physical assumptions and build simplified recipes to model the different processes taking place. Models usually use a considerable number of free parameters to adjust some poorly known processes, provide good statistical information of galaxies and their properties and generally require less computing resources. The latter, n-body/SPH simulations, implement some fundamental interactions, fill the simulation volume with some initial configuration and let the system evolve over time. This approach typically suffers from shortage of computing resources, for they are way more demanding in this sense and, additionally, most relevant processes involving baryon physics occur at sub-resolution scales making it inevitable to introduce recipes borrowed from models that come bundled with free parameters. However, it has been proved that they are very good at reproducing the dynamics of collisionless fluids such as the clustering of dark matter and they render specific information on the actual spatial distribution of the systems.

On the one hand, semi-analytic models usually build a merger tree that contains the history of halo formation through mergers and accretion with the aid of pure gravitational N-body simulations or Monte Carlo methods based on the extended Press-Schechter formalism and others [HNS$^+$02]. The use of numerical simulations in this very first stage is computationally more demanding but has proved very effective and accurate and is the latest tendency in the field. From here on, models try to develop and adapt recipes to account for all the physical processes involved, such as the radiative cooling, the star formation, the energy and chemical feedback provided by evolved stars, etc.

On the other hand, simulations use N-body codes to treat the collisionless gravitationally interacting component and Smoothed-Particle Hydrodynamics (SPH) to account for the dynamics of the gas component. Typically, a cosmological simulation volume is chosen, a time step is set and the initial conditions are specified before one lights the wick of the actual run. Obviously, cosmological simulations are severely limited by both their spatial and temporal resolution, so that all the processes taking place below this threshold are considered sub-resolution and must be dealt with using recipes.

Thus, we can say that both approaches have advantages and drawbacks and they should complement each other at the time of drawing results on galaxy formation and evolution processes.

The present work, though, is particularly interested in semi-analytic models, and especially in the Analytic Model of Inter-galactic medium and GAlaxy evolution (also known as AMIGA), developed by the group led by Dr. Eduard Salvador, which is designed specifically to track the coupled evolution of galaxies and IGM since the first stages of the Universe. The model presumes to be fully analytic, for it neither relies on N-body simulations nor Monte Carlo methods to build merger trees, but on interpolation techniques applied to a grid of halo properties. In this fashion, AMIGA also models molecular cooling and Population-III stars and the massive black holes they are thought to have produced at the end of their lives, identifying them as key players in the galaxy formation and evolution process. The model has been accurately tuned to dismiss any unnecessary free parameter, and strives to causally connect the larger number of processes possible to achieve the maximum self-consistency.

1.1 Work aims and tasks

Sometimes a great effort is put in the development and debugging of a software system that produces great results, leaving the part where these results are made available to the community somewhat unattended. It is the aim of this work to provide AMIGA with a lightweight, accessible and user friendly web front end to make the output of the model available to the scientific community and the world in an elegant and yet powerful manner. This piece of software is to be the visual facade of the powerful software system that lies behind and thus an important module that must be addressed with care.

The AMIGA front end, called *AMIGA High-z Observatory*, is classified within the virtual observatories family. Virtual observatories are pieces of software that, in astronomy, act as bridges between the raw user and an astronomical database, allowing the observer to query data on objects and structures. Usually, these systems merely bypass queries to a distributed database and display raw or formatted results -such as star listings-. Our case, however, is somehow different. The model is not backed by a database but uses files as its main output channel. Additionally, there exists a command-line program, the *look*, aimed at producing comprehensive output information (plots, histograms and formatted data) from the data files generated by AMIGA. This program, although suitable for internal use and as a development debugger, does not fulfill the necessary requirements to be employed as a virtual observatory. It requires a considerable expertise on the system to make it work properly, it can't be used concurrently by more than one user at the same time (given it could be published in some standard and accessible environment), its command-line nature makes it highly dependant on the operating system and hardware architecture in use, it does not comply with the standard usability guidelines and, most important, it has not been optimized to offer swift responses to requests, regarding data management.

This void is intended to be covered by the AMIGA High-z Observatory, whose requirements are the following.

1.1.1 Functional requirements

As defined by requirements engineering, functional requirements state specific behaviours of a system. They define the internal workings of the software, that is, the calculations, technical details, data manipulation and processing, and other specific functionality that show how the use cases are to be satisfied. We shan't be that scrupulous in this document, giving just a descriptive view that accounts for the workings of the system as a whole, and not detailed descriptions for each function.

- The system functionality must be divided in logic units of computation called items, each of which is in charge of processing and displaying information on concrete properties of galaxies/haloes/IGM.
- Each item will proceed as follows:
  1. First it must gather some information on the observation the user is interested in, called parameters -such as redshift of observation, narrow or broad band, filter, type of galaxy environment, etc.-. This and the following steps are optional, for some items, such as the *Redshift vs Cosmic Time*, do not need input parameters to produce results.
  2. These parameters are then validated so that their values are within an acceptable range. Each parameter defines its own value range, which may be continuous or discrete.
  3. Once parameters (if any) are validated and correct, the system must display the specific information defined by the item, which will be one or more standard plots, histograms or processed data.
  4. The user must have the option to download the processed data as comma-separated values (CSV).
- The system must offer a standard procedure to submit feedback to the group, either on the system itself or on the results it outputs.

1.1.2 Non-functional requirements

Non-functional requirements, in contrast, specify criteria that can be used to judge the operation of a system, rather than specific behaviours. An initial list of non-functional requirements to be fulfilled by the system is the following.

- The system must be accessible by users through a web interface that accomplishes the standards defined by the W3C: CSS, HTML, XML.
- The client web application must be as lightweight as possible, leaving the heavyload of the processing to the server that will be maintained and managed by the group.
- The client web application must avoid page reloads as long as possible, and it must be designed as if it were a regular desktop application.
- The client web application must comply the basic standards on usability and graphical design defined by ISO9241 parts 11, 12 and 13 [ISO06].
- The client web application must perform within acceptable limits in low-end devices such as aged computers or high-end mobile handholds.
- The server-side application must always reach response times no longer than 10 seconds in the worst-case scenario. This is the request-response time from the client's point of view. Any actions necessary must be taken in the server to ensure this requirement, such as data caching.
- The server-side application must be capable of running in a desktop-standard environment and must be able to serve several requests at the same time. It must implement concurrent data access.

1.2 Similar work

As it has previously been stated, most of, if not all, current virtual observatory programs are completely database oriented, serving as a mere facade to generate SQL queries against a database and serve its results. This is the case, for example, of the US National Virtual Observatory [NVO] or the European Virtual Observatory [EVO], which comprises several catalogs and databases such as the Aladin Sky Atlas [FBBO], the Sky View [SBNM] or the DataScope [VAO]. Some of them also offer, if available, image data on the sources fetched. Both are huge projects that join efforts of several scientific communities and research groups.

Chapter 2

Application design and implementation

The AMIGA High-z Observatory is structurally divided into two main components, the model and the observatory. The model is in charge of managing the data (loading and writing of files, caching, compressing and providing the API[1] to access the data) and the observatory implements the web application, which in turn is divided into the client side and the server side. The server side of the observatory deals with the model component to fetch and prepare the data for their display in the client side, that is executed in the client's browser.

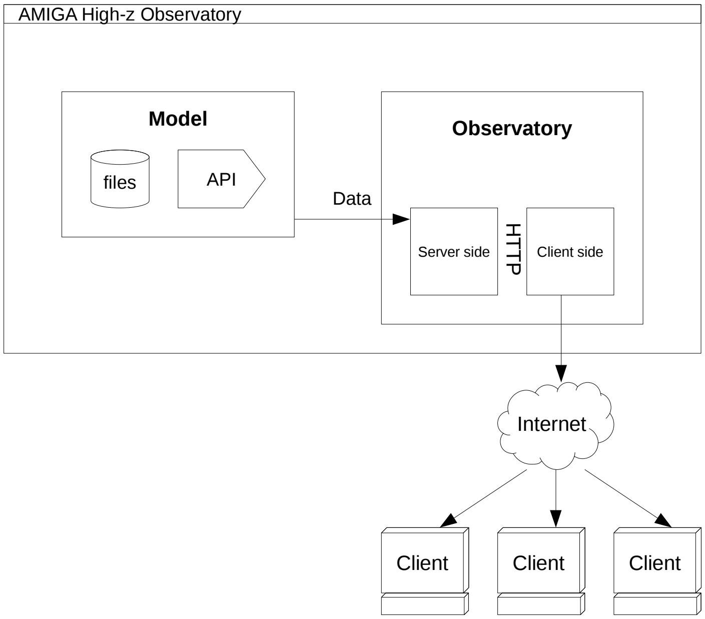


Figure 2.1: Overview of the main components that take part in the AMIGA High-z Observatory.

[1] Application Programming Interface, a particular set of rules and directives computer programs can follow to communicate to each other

The two modules have been designed to work independently from each other, using as a connection a couple of classes[2] called `AMHolder` and `AM` (from Amiga Model), which are declared and implemented in the model component and used in the server part of the observatory component to load the files needed to process each request. Hence the model can be used by any other program -such as, for instance, a standalone desktop application or the *look* itself, given a FORTRAN API were provided-.

In the forthcoming sections we shall discuss the design approaches adopted to implement these two modules and we will give a bird's eye view on their internal working.

2.1 The data model component

The model component can be pictured as an abstraction layer over the data files that must be used in the observatory to produce comprehensive results. This abstraction layer encapsulates and simplifies all the operations that deal with the data such as data loading, access, compression and caching.

Please, note that when using the *model* word in this context we are overloading the language, referring to the data logic rather than a semi-analytic model.

2.1.1 Structure of the data

Due to the nature of AMIGA, the source data with which the AMIGA High-z Observatory must work have some peculiarities. AMIGA is run with a set of input and output parameters. The input parameters are the usual free parameters we discussed in chapter 1, which are present in any semi-analytic model and account for poorly known physical processes. On the other hand, output parameters establish cutoff properties that restrict the span of results that can be extracted from an execution. For instance, the filter type (Johnson, SDSS or narrow band), the frame location (observer or galaxy) or the redshift (z) of the sample are output parameters.

Consequently, we have different groups of files (which we call *sets*) outputted by AMIGA corresponding each to a different set of output parameters. In this style, we are able to load the same files for each set, whose value depends on a couple of parameters provided by the user. However, since the model component intends to provide an abstraction layer over the actual data, the mechanism to load and select a file from one or another set is to be transparent to the user, in this case the observatory component.

To sum up, our data is separated into sets of files, containing each set the same files with different data, and these must be loaded and maintained into memory in a transparent, automatic manner.

[2] In the object-oriented paradigm, a class represents an entity, a blueprint from which individual objects are created. In object-based programming, a class is simply a source file holding functions, procedures and variables.

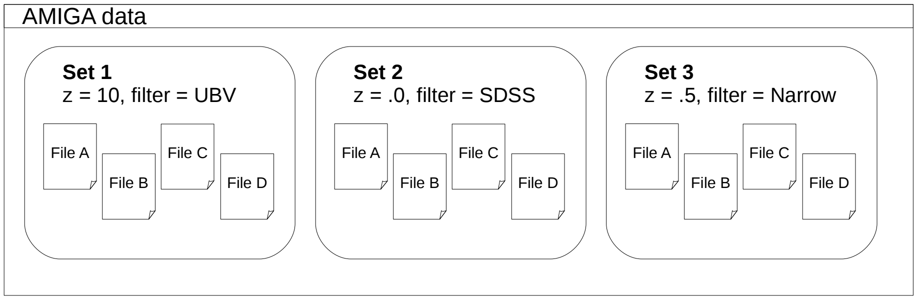


Figure 2.2: Organisation of data files in AMIGA High-z Observatory.

Physically, set files are separated into folders, each named with the set id, which is a natural number arbitrarily assigned. Logically, set ids and properties are specified in an `xml` file that is parsed and loaded at the beginning. This file is called `metadata.xml` and must be put in the root of the source folder. Once parsed, the meta data is to help us load the file from the correct set.

Down to a lower level, the `AM` class contains all the fields, variables and matrices, that are to become the holders in memory of the data stored in AMIGA files when the observatory is running. However, the files are not all loaded at once for performance reasons, but in an on-demand basis. When the system is booted, the meta data is parsed and the initializer creates as many instances of `AM` as sets are defined. These instances are mere void shells, containers whose vectors and matrices are allocated and loaded when needed. This way, we minimize memory usage and improve the response time. Additionally, we have set up a cache layer that loads and unloads data depending on current memory usage. The maximum size of the cache can be specified in a properties file called `model.properties`, which also defines the base location of the data folder. We shall talk more about the cache layer in section 2.1.5.

2.1.2 File compression and handling

Files coming from AMIGA are produced by the *look*, which is a command-line console-based program that displays results from AMIGA. The *look* is a FORTRAN-based application and it outputs formatted files aligning values in columns and keeping the amount of characters used to represent each value constant. This leads to the addition of many white space characters that take disk space. Additionally, most AMIGA files have an enormous quantity of zero values written in scientific notation (0.0000000E+00), taking, for example, 13 ASCII[3] bytes to represent each zero.

All these facts result in file sizes well over the gigabyte (up to 6 GB in some particular cases), sizes totally unsuitable for a concurrent-access system. To solve this issue we designed a pre-process stage where files are shrunk and compressed in a way that they can still be read directly without need for decompression (this is, keeping the read time the same). Basically, the pre-process eliminates redundant white spaces and takes advantage of the fact taht most of the time we are dealing with

[3] American Standard Code for Information Interchange, is a character-encoding scheme based on the ordering of the English alphabet.

sparse matrices pretty filled with zeroes, shrinking successive zeroes into just two values, the marker 0 and the number of zeroes to insert in the position.

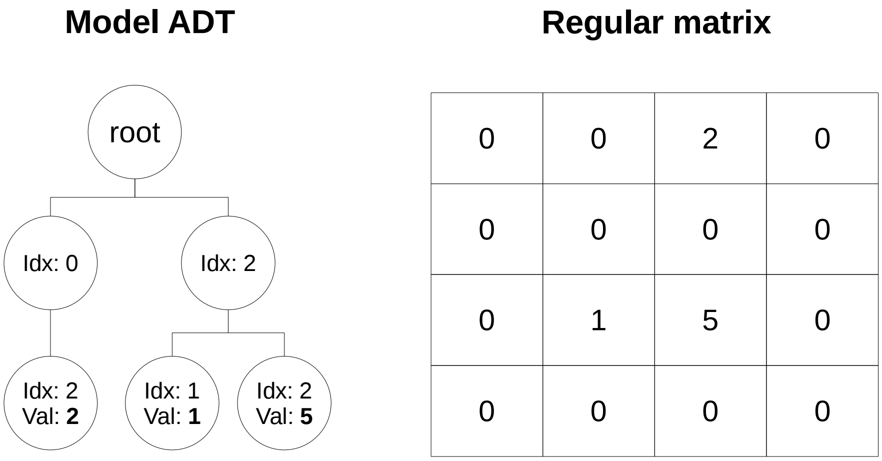
This pre-process decreases, for instance, the `alba.look` file size from 1.8 GB to only 150 KB. This, obviously, does not work that well with all files, for it strongly depends on the data, but in general we reached file size reductions of more than 70% in the majority of cases.

The pre-process is designed as a command-line application that processes a whole set of files at once and writes it into a desired folder. This pre-process is only to be executed once per file set.

2.1.3 New ADT for sparse matrices

Most data contained in files is to be loaded into memory in the form of matrices. This is, however, a naive approach in some cases where most matrix positions are filled with zeroes (sparse matrices) that occupy useless space in memory when the application is running and whose dimensions are far too large to be looped over in a reasonable time. If the matrix data type is floating point value (float) or integer value, each zero takes 4 bytes. If it is a double precision floating point value (double) or long integer value (long), each zero takes 8 bytes. Multiply this by the number of zeroes and we have a huge waste of memory and access time.

In order to minimize data access time and memory usage, I designed a new Abstract Data Type (ADT) to store sparse matrices in an efficient way, in terms of both memory usage and access speed. The structure behind this is a tree whose nodes hold at each level the current dimension index, from the root right down to the leaves, where actual values are stored. Thus, we can get the indexes for a value in $\sim O(d)$, being d the number of dimensions. Also, the access time in this ADT is $\sim O(d \cdot log(m))$,where m is the mean dimension size.



Model ADT

Regular matrix

0	0	2	0
0	0	0	0
0	1	5	0
0	0	0	0

Figure 2.3: Comparison between the ADT and a sparse matrix for the same content.

The ADT comes with the necessary operations to create, modify and access the data type in a transparent manner. It is obvious that this structure uses much less memory than a regular matrix (sparse), specially when the matrix dimension grows. Also, if we want to loop over the structure, the matrix is much slower than the ADT. However, the indexed access, which is immediate in a matrix, takes a bit longer in the ADT. In our case we will not be using indexed access, but loop sequential access, so it is not a big deal.

2.1.4 Model loader and processor

Model files are loaded into memory by the model loader. It uses a group of reflection techniques[4] to fetch and load individual files in an elegant and yet powerful way.

The process of loading a couple of model fields proceeds schematically as follows: the API class `AMHolder` receives the call to load the set of fields. This call gets the redshift and the filter as parameters, along with a list of all the fields to load. First, the loader uses the redshift and the filter to select the most appropriate model set using the meta data available. Then, it checks the correspondences field-file (which hold information of which fields stored are in which files) using previously loaded data from a file called `correspondences.xml`, and discovers which are the files to load. Once it gets the files, it just delegates the work to a component called reader, which is in charge of actually opening, reading and loading files into memory entities such as variables and matrices. Since in the current implementation there is no direct correspondence between files and fields, the reader contains a function for each file that, using an abstract file reader, gets the data from the actual file and loads it into the correct memory field in the correct order. Additionally, there is an allocate function for each file that allocates the memory space required for the fields stored in this file. This method is called right before the loader method. Finally, there is also an unload method for each file, which is called by the cache manager when it needs to free resources. The cache is covered in the following subsection.

2.1.5 The model cache

The model component is equipped with a cache layer, which maintains loaded data into memory and ensures the number of bytes loaded at a time does not exceed a certain amount specified by the system administrator in the `model.properties` file. To achieve this, each field has an operation which returns the number of bytes it occupies. In the case of regular matrices this operation is very straightforward (simple multiplication of capacity by data type size). In the case of the ADT, however, we need to add an additional attribute in each node maintaining the current number of values below that node, so that for any node this attribute is calculated by summing the values of the same attribute of its children. In the case of leaves, the attribute value is one. This can be maintained very easily updating the values in the insert operation as we depth-first search the tree to add a new leaf.

[4] Reflection in computing science is the process by which a computer program can observe (do type introspection) and modify its own structure and behavior at runtime.

The structure of the cache itself is implemented with a heap[5], where the keys are the time since the last access to the element, making the elements that have not been accessed for a long time the first eligible to be unloaded.

This system offers a pretty good performance, yielding a $\sim O(log(n))$ for the insert operation (n being the number of elements in the heap) and a $\sim O(1)$ for the polls.

2.1.6 Statistics of the model component

The model component consists, at the moment, of 3737 lines of code distributed in 24 classes organized into 7 packages. There are also 3 configuration files (`correspondences.xml`, `metadata.xml` and `model.properties`). All this is packed into a project which defines a library descriptor, used to create a library file that is to be used in the web application in a transparent manner, as we shall see in the next section.

2.2 The observatory component

The observatory component is even more complex in its design than the model component. Because of its client-server nature, intrinsic to any web application, it requires work in very different areas, such as GUI[6] design, usability, software engineering or scripting. In order to ensemble all these pieces together, there are some frameworks available to make the developer's life easier.

2.2.1 Web frameworks

Most classical (now quite obsolete) frameworks based on servlets implemented the Java Servlets request-response paradigm with actions and forms. This resulted in almost static web pages that needed to reload every time the server was to be contacted (even though this could be solved using AJAX[7] techniques that had to be programmed by hand). Nowadays, modern web frameworks tend to lead the change to an asynchronous client-server infrastructure, where the client browser downloads and executes locally the application component (usually written in JavaScript[8]), which in turn communicates with the server-side via asynchronous HTTP[9] calls using AJAX.

In the latter group we find GWT[10] [Goo], which goes one step beyond. GWT allows the developer to write the web application almost entirely in pure Java as if it were a desktop application, that is later compiled into a web application (HTML[11] + JavaScript + pure Java) and deployed into a

[5] A heap is a specialized tree-based data structure that satisfies the heap property: if B is a child node of A, then $key(A) \geq key(B)$

[6] Graphical User Interface

[7] From Asynchronous JAvascript and Xml, is a group of interrelated web development methods used on the client-side to create interactive web applications.

[8] Or Client-side JavaScript, which is a scripting language weakly typed implemented as a part of a web browser.

[9] HyperText Transfer Protocol, the foundation of data communication for the World Wide Web.

[10] Google Web Toolkit, Google's web application framework, very popular among web developers lately.

[11] Stands for HyperText Markup Language, the standard language for web pages.

servlet container[12] in an old-fashioned way. This is the most suitable option for our AMIGA High-z Observatory system for the following reasons:

1. Once client side has loaded the application, it does not require browser reloads, giving the impression of working with a standalone desktop application, offering a better user experience and higher responsiveness.
2. The application is organized as if it were a pure Java application, avoiding the folder structure and meta-data required by Java EE[13] applications. This would increase the maintenance cost, specially in a scientific environment where one can not usually find specialists versed in such technologies.
3. Avoid JavaScript development, a recognized source of browser-dependant errors and bugs. GWT infrastructure provides cross-browser support via built-in widgets and composites that come bundled with specific implementations for each popular browser[14], bridging the problem of browser compatibility.
4. Finally, GWT's compiler offers the option to automatically obfuscate the produced code, feature very suitable to hide the actual scripting code downloaded to the client.

2.2.2 Structure of the solution

The structure of the observatory is very intuitive, if one thinks of the basic building blocks. There are two components, the server and the client, linked by the `Controller`, which is the class in charge of submitting asynchronous requests to the server from the client, gathering responses (either error or success responses) and passing them down to the client again to display the results.

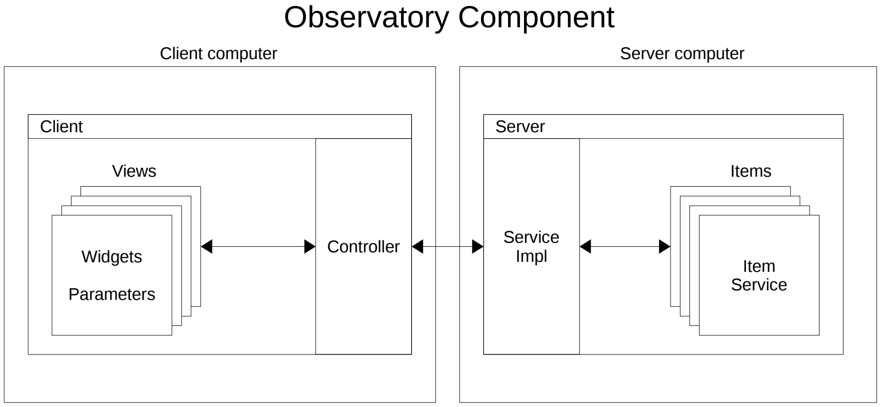


Figure 2.4: Schematic structure of the observatory component.

Without entering into too much detail, the information flow goes like this: The user selects the

[12] HTTP server that implements the web component contract of the Java EE architecture, including security, concurrency, life cycle management, transaction, deployment, and other services

[13] Java Enterprise Edition, which sets the standard for Servlet-based web applications

[14] Namely, IE6-9, Google Chrome, Mozilla Firefox, Apple's Safari and Opera.

item to display from the menu and enters the value of the required parameters in the view. Then the view constructs a map of pairs `[name-value]` to represent the parameters and sends it to the controller, which constructs the `ItemRequest` object and sends the asynchronous call to the server. The `ItemRequest` is simply a DTO[15] that carries the parameters and the token of the item to be executed. The controller also defines couple of functions that contain the logic to be executed if the call completes normally or fails, respectively. Then, the server stub receives the request, automatically fetches the entity that will process it and delegates the work to it. Once the request is processed, the result is sent back to the client via the `ItemResponse` object, which arrives to the controller, that sends it back to the view to display the appropriate results.

In the next subsections we shall inspect the structure and building blocks of the observatory component more carefully.

2.2.3 Client component

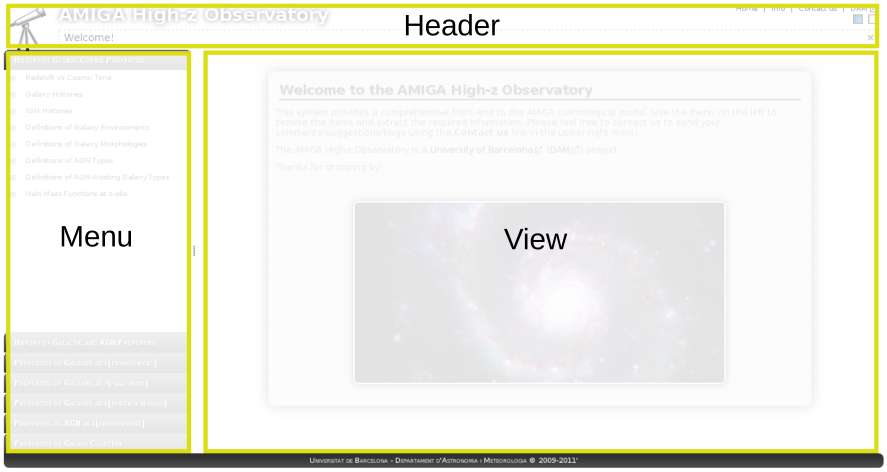
The client component is ultimately written in HTML and JavaScript and is completely downloaded to the client's browser when it loads the AMIGA High-z Observatory web application. The client is executed locally in the client's computer and communicates with the server via HTTP requests/responses. However, with the use of GWT, the source is written in pure Java and then compiled into JavaScript, so that the architecture resembles more that of a standalone application.

The client component can also be seen as a graphical interface to functionality offered by the server, that is, a visual facade to access data produced by AMIGA. In this fashion, the client is just a mere view component.

2.2.3.1 Layout

Graphically, the client component is laid out following an immutable structure pictured in image 2.5. There's a common header that displays the title and a set of links to the home, feedback, information and Departament d'Astronomia i Meteorologia pages. The header also holds the bread crumbs, a small informative panel that shows the item we are browsing and its parents. Then there's a left menu that gives access to the various items available and a right container where views are rendered. Actual content is displayed in there.

[15] Data Transfer Object, used to transfer information


AMIGA High-z Observatory
Home | Info | Contact us | DAM
Welcome!
Header
History of Global Cosmic Properties
Redshift vs Cosmic Time
Galaxy Histories
IGM Histories
Definitions of Galaxy Environments
Definitions of Galaxy Morphologies
Definitions of AGN Types
Definitions of AGN-Hosting Galaxy Types
Halo Mass Functions at z-obs
Menu
History of Galactic and AGN Properties
Properties of Galaxies as f(environment)
Properties of Galaxies as f(halo mass)
Properties of Galaxies as f(position in halo)
Properties of AGN as f(environment)
Properties of Galaxy Clusters
View
Universitat de Barcelona - Departament d'Astronomia i Meteorologia © 2009-2011'

Welcome to the AMIGA High-z Observatory

This system provides a comprehensive front-end to the AMIGA cosmological model. Use the menu on the left to browse the items and extract the required information. Please feel free to contact us to send your comments/suggestions/bugs using the **Contact us** link in the upper-right menu.

The AMIGA High-z Observatory is a University of Barcelona (DAM) project.

Thanks for dropping by!

Figure 2.5: Graphical layout of the AMIGA High-z Observatory.

2.2.3.2 Application manager

The **application manager** is a component that creates and initializes the views from tokens, usually in response to some event occurred in the application, such as a click on a menu item or a link, or a browser reload. Tokens are strings that unequivocally identify views and are used to convey information on view load requests. The menu, in its turn, is a graphical component that displays the set of available items and generates the token to send to the application manager for its instantiation. Once the application manager has instantiated a view, the view itself takes over its own life cycle and is in charge of displaying and gathering the information it is programmed for, until it is replaced by another view.

2.2.3.3 Views

Obviously, the most important part of the client are the **views**, represented by the `View` class, which is at the top of the inheritance[16] tree and is extended[17] by all the actual entities that control the elements that are displaying in the application.

The graphical aspect of views can be defined using XML markup with GWT's `UiBinder`. It allows to define the visuals of the application in a very similar way as one would do for a regular web application, using CSS and HTML. View files have the extension `.ui.xml` and are located in the same package as the source view file. We shall talk more about the definition of views in XML files in subsection 2.2.3.4.

There are two kind of views: Static views and Item views.

[16] In object-oriented programming, inheritance is a way to compartmentalize and reuse code by creating collections of attributes and behaviours called objects which can be based on previously created objects.

[17] In an inheritance structure, if an object B extends another object A, B automatically acquires A's protected and public behaviours and attributes, as well as those of the objects A is extending.

On the one hand, **static views** are used to display fixed, logicless pages such as the information page, which displays information about the system, the feedback page, which provides a feedback form that allows users to send back comments and bugs on the system, or the front page, which displays at start-up. Static views do not extend the `StaticView` class, but rather populate it with a page, which defines the graphical aspects of the view. It is in this sense that static views resemble static HTML pages.

On the other hand, **item views** contain most of the logic of the client component. The functions offered by the AMIGA High-z Observatory, as well as those in the *look*, are organized into items. Each item is in charge of producing comprehensive results in the form of plots, text or structured data, referring to a property of galaxies or AGN. Thus, each item view needs to collect a set of parameters representing the user's needs and display the results. Therefore, item views are composed of the parameters view, which renders at the top, and the results view, which renders below it.

2.2.3.4 Widgets and parameters

The `ItemParams` class is a container for parameters, which are abstract entities based on concrete widgets. Our widgets, technically, are an extension of GWT's widget interface. A widget, logically, is an independent, reusable box of functionality that has a graphical side and a logical side. The widgets can be laid out into views using predefined XML tags in the XML view definitions. Also, using layout panels (which are also widgets), one can distribute and arrange widgets visually within views in an easy way through simple XLM markup tags in `UiBinder`. Actually, views are no more than container widgets made of smaller widgets that are displayed in the content placeholder of the layout.

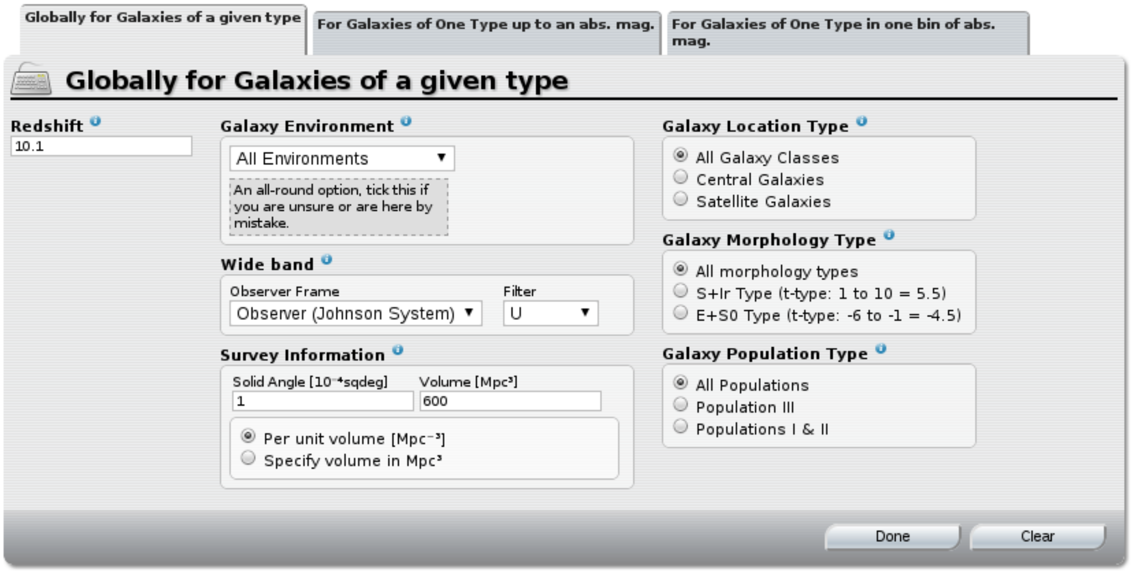
The typical parameter XML definition looks like this:

Listing 2.1: `UiBinder` definition for the parameters box of item 2.1, *Galaxy stellar mass funtion.*

```
<ui:UiBinder xmlns:ui="[...].gwt.uibinder">
   <g:HTMLPanel>
    <vo:DivHorizontalPanel>

     <vo:DivVerticalPanel>
      <vo:FormField description="The observation Redshift (z)">
       <g:Label>Redshift</g:Label>
        <vo:RedshiftSelect ui:field="redshiftSelect"></vo:RedshiftSelect>
      </vo:FormField>
     </vo:DivVerticalPanel>

     <vo:DivVerticalPanel>
      <vo:FormField description="The solid angle and volume of the survey">
       <g:Label>Survey Information</g:Label>
        <vo:VolumeLite ui:field="volumeLite"></vo:VolumeLite>
      </vo:FormField>
     </vo:DivVerticalPanel>

   </vo:DivHorizontalPanel>

   <g:HTMLPanel styleName="ButtonContainer">
  <g:Button ui:field="done">Done</g:Button>
  <g:Button ui:field="clear">Clear</g:Button>
   </g:HTMLPanel>

  </g:HTMLPanel>
</ui:UiBinder>
```

As you can see, this parameters view contains a horizontal panel which in turn contains two vertical panels inside, where the parameters are located. the `FormField` tags are parameter wrappers used to add the label and description to all parameters in a unified way. At the bottom of the file there is the button container, which holds buttons for both submitting and clearing the parameters. The submit button has a click handler in the view, which, when triggered, validates the parameters, puts their value in a map and connects directly with the controller, who sends the call to the server.

This is what a typical parameters box looks like:


Globally for Galaxies of a given type
For Galaxies of One Type up to an abs. mag.
For Galaxies of One Type in one bin of abs. mag.
Globally for Galaxies of a given type
Done
Clear

Redshift: 10.1

Galaxy Environment: All Environments ▼

An all-round option, tick this if you are unsure or are here by mistake.

Wide band
Observer Frame: Observer (Johnson System) ▼
Filter: U ▼

Survey Information
Solid Angle [10⁻⁴sqdeg]: 1
Volume [Mpc³]: 600
● Per unit volume [Mpc⁻³]
○ Specify volume in Mpc³

Galaxy Location Type
● All Galaxy Classes
○ Central Galaxies
○ Satellite Galaxies

Galaxy Morphology Type
● All morphology types
○ S+Ir Type (t-type: 1 to 10 = 5.5)
○ E+S0 Type (t-type: -6 to -1 = -4.5)

Galaxy Population Type
● All Populations
○ Population III
○ Populations I & II

Figure 2.6: Graphical aspect of parameters box of item 3.3.1, *M/L Histograms*.

For instance, in figure 2.6 one can see the following 7 parameters:

Redshift It is a simple text box whose value must be a floating point number within the certain range [0..64].

Galaxy Environment It is a fully customized parameter consisting of a drop-down box with the value options and an informative box that offers a description of the currently selected entry.

Wide Band Also, a fully customized parameter with two drop-down boxes, the first selects the frame (Observer or galaxy), and the second selects the actual broad-band filter. Note the contents of the second box change as the value of the first change, displaying the UBV set of filters for the observer frame and the SDSS set for a galaxy frame.

Survey Information In this parameter the user must submit the solid angle and the volume of observation, as well as the format in which he desires the results to be shown.

Galaxy Location Type It is a simple radio-button box with multiple options for the galaxy location.

Galaxy Morphology Type Just like in the previous case, it is a radio-button box with multiple options.

Galaxy Population Type Same as the previous two.

Bear in mind that this is just a small taste, and that the span of parameters is way larger than that shown here.

All parameters are widgets, but not all widgets are parameters. For example, there are widgets that contain other widgets, such as vertical or horizontal layout panels. Also, there are abstract parameter widgets that define a general looks and structure and that are extended and made tangible by actual parameters used in views. Some of the abstract parameter widgets are the *check box group*, the *radio button group*, the *text input* and some more complex ones, such as the *double combination picker* or the *interval picker*. Concrete parameters are, for instance, the *photometric band*, the *galaxy location type*, the *color-color picker* or the *halo mass*.

All these parameters are grouped in sets depending on the item needs, and displayed by the item view when required. Additionally, all parameters implement the `Parameter` interface, which defines useful functions (such as `getName()`, that gets the parameter name, `getValues()`, that gets the value/s, or `initFromCookies()`, which attempts to initialize the parameter value from the browser cookies) to treat all parameters in a unified, single manner forgetting about the concrete type. This adds a great simplicity and structure to the code. Moreover, parameters also implement the `validate()` function, which is called for each one of them right before the call to the server and returns a list of all the validation errors encountered for the parameter. If the mentioned list is empty, the field value is correct and therefore the parameter passed the validation. An error message is displayed if any of the parameters does not pass the validation, requiring the user to enter a correct value. The actual validation is carried out by the `Validator` module, giving a unified, generic solution to validate numbers and strings.

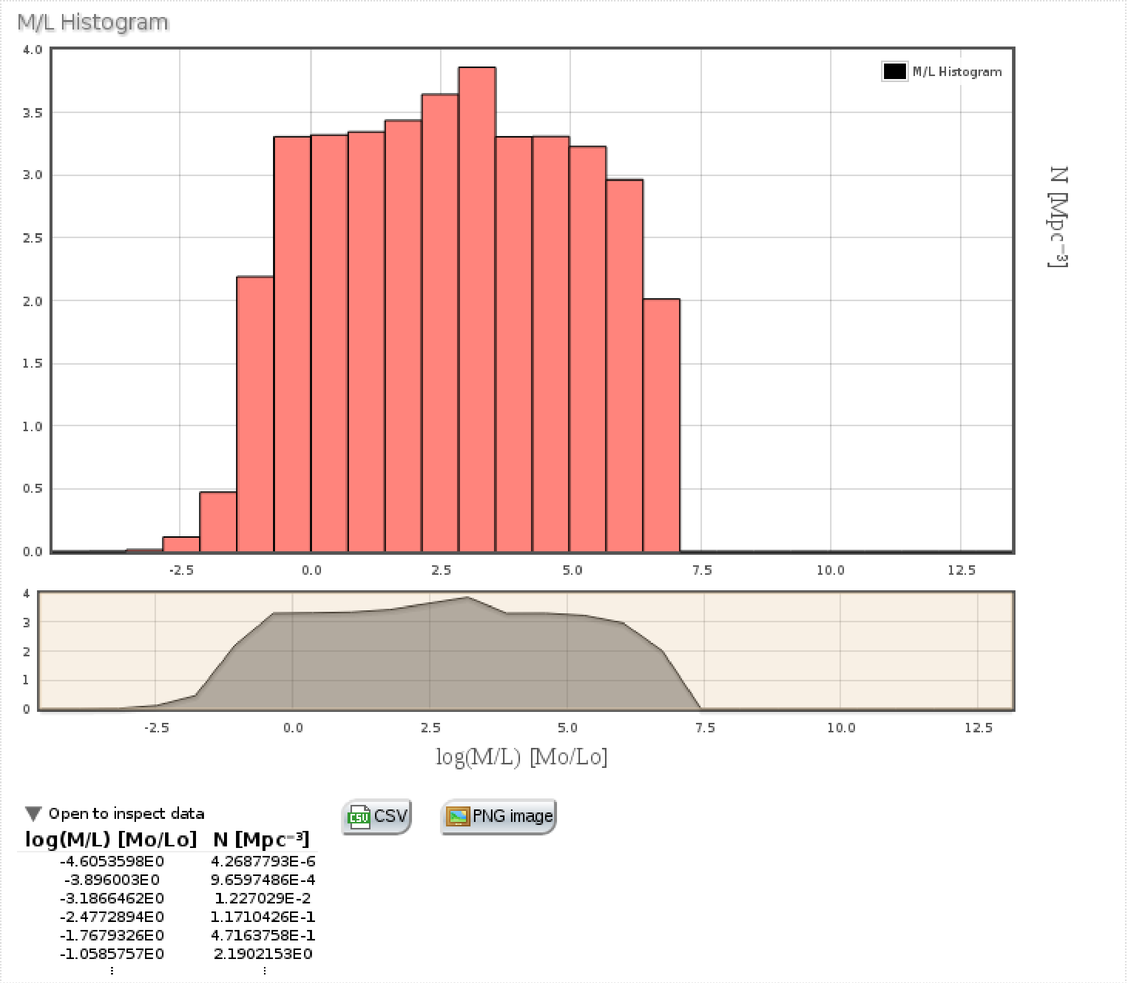
2.2.3.5 Displaying results

When the `ItemRequest` is sent to the server and processed, it generates a new DTO called `ItemResponse`, which is designed to convey the results back to the client. The `ItemResponse` can carry plot data, tables, messages and also the state of the response (which indicates whether the processing succeeded or failed). When the controller receives the asynchronous response, it checks for errors. If there are, it tells the current item view to display them. If everything went well, it handles the response to the item view and tells it to display the results accordingly.

There is no specific, dedicated way to handle the result display for each item, but rather the process is the same for all items and depends strongly on the `ItemResponse` contents. This means that there's just one implementation of the method that displays the results, which receives the response and creates the actual output such as plots and tables in the current item view launching the method.

If the `ItemResponse` contains plots or histograms, the item view shall prepare the Flot plugin[18] [Lau07] and take the necessary actions to set up, lay out and display the plots. Also, it sets some other structures, such as the tabular data into a disclosure panel, or the button to download the data as a CSV file[19]. If the response contains messages the item view shall display them accordingly. The flot plugin plugs into GWT via the GFlot library [Leo09], and some adaptations and customizations have been necessary to make it suitable for our project. For example, we had to add support for different line patterns, observation data and plot image capture.

Let's see a typical results view.



log(M/L) [Mo/Lo]	N [Mpc^{-3}]
-4.6053598E0	4.2687793E-6
-3.896003E0	9.6597486E-4
-3.1866462E0	1.227029E-2
-2.4772894E0	1.1710426E-1
-1.7679326E0	4.7163758E-1
-1.0585757E0	2.1902153E0
⋮	⋮

Figure 2.7: Graphical aspect of the results of item 3.3.1, *M/L Histograms*.

[18] Flot is a pure JavaScript plotting library for jQuery, a JavaScript language extension.

[19] Coma-Separated Values, standard for data sheet files such as XLS.

As one can see in figure 2.7, there is a main source of information, the histogram (which could be a line plot, or even multiple plots). The plot usually displays the labels, a legend, the raw data in a table (contained in a disclosure panel that can be expanded), a button to download the plot image and a button to download the data in CSV format. There is also an overview below the main plot which lets the user select an x-axis range to display in the main plot.

Finally, it is worth a mention to state that the results are not displayed in place of the parameters, but rather below them, so that the user can change their value and launch the processing again. A movable boundary is provided to adjust the size of the results view.

2.2.4 Server component

Luckily, the server component of the observatory is simpler than the client in its structure, but it contains the *item services* that actually implement the processing of data and construct the responses.

In the server side, there are two basic services that await calls. The first is the AMIGA service implementation (AmigaObservatoryServiceImpl), which attends the requests regarding items. The second is the view service (ViewService) and is in charge of small units of processing and data gathering where view (client) elements (usually parameters) require of some server resources. For example, the *mass range* parameter, intended to pick a range of masses, fetches its values from the tgm matrix of the model and these depend on the redshift and band selected at the moment (for redshift and band are used to select a set, implying a different tgm file to be loaded into the matrix). This second set of services offered by the server component contains rather simple logic and is of few interest for the present document.

Turning our heads back to the AMIGA service implementation, there is an item service for each item, that provides methods to transparently get the values of the parameters from the DTO ItemRequest. The BaseService abstract class must be extended by all item services and it, in turn, extends the AMHolder from the model to give the items access to the model attributes and fields. The value of the reference to the model contained in the AMHolder shall change when a new set of files is requested to load. Additionally, all item services implement the ServiceInterface interface, designed to be able to treat all service executions in a unified, transparent way. Basically, this interface defines the ItemResponse process(ItemRequest request) method, which must be implemented in all item services. Then, when the observatory service implementation gets the call from the client with the request, the token is parsed to extract the item service class name. Then, the class is reflectively instantiated into a ServiceInterface object, right before the process method is called with the ItemRequest got from the client.

The first thing that the actual item service does when it is executed is to fetch the data fields it needs from the model. To do so, one gets the filter and redshift from the request, prepares an array with all the necessary fields and calls the model API to load them. The set from which to load is selected transparently and the call returns an success/error state used to inform the client.

Let's see a code extract that does the trick.

Listing 2.2: Usual data loading code snippet.

```
String filter = getPhotobandType(params);
double z = params.getDoubleParameter(Constants.PARAM_REDSHIFT);
data = new String[] { "oz", "zmax", "alrcom", "ls0", "clusterhalo"};
if (!load(filter, z, 1d, data)) {
   return getErrorLoad();
}
```

In listing 2.2, the strings in red are the names of the fields to load. The filter and z variables are used to select the appropriate set. The call to the `load` function returns a boolean which, if false, indicates there has been a problem and thus we need to return an error response to the client. Otherwise, if it is true, the data has been successfully loaded and is automatically available to the item through the `AMHolder`, whose execution proceeds normally.

Once the data is loaded, the item proceeds in a sequential manner. Most of the items are mere translations from FORTRAN to Java, using the *look* program as a source. Usually, the local variables and output arrays are initialized in the first block, then some model matrices are iterated over and the output arrays are populated in the process, and finally, the response object (`ItemResponse`) is populated and sent back.

However, there is a special group of items whose implementation diverts from that in the *look*. These items' execution is driven by the data structure, and all of them use data from the new tree ADT for sparse matrices pointed out in section 2.1.3. These items use a function defined in the `AmigaService` class (a class that extends `BaseService` and adds some functionality useful to certain items) that iterates a given instance of the ADT in a generic manner and executes a function for each value found, whose implementation must be provided by the item. This means, basically, that the item processing triggers when a new value is found in the sparse matrix ADT, and then processed elegantly. This shrinks the code in each item dramatically, for all the iterating and looping takes place in a super-class (parent) and is only written once, leaving to the item the actual processing of values.

There is an example of an actual processing below.

Listing 2.3: Service code snippet for item 4-2-1, *Disc central surface brightness histogram*

```
gotValue(Float value, Stack<Integer> indexStack) {
  //Prepare indexes
  int iy = indexStack.pop();
  int ib = indexStack.pop();
  int bs = indexStack.pop();
  int b = indexStack.pop();
  int lt = indexStack.pop();
  int ie = indexStack.pop();
  int sd = indexStack.pop();
  int gt = indexStack.pop();
  int g = indexStack.pop();
  int m = indexStack.pop();
  ie = 1;
  k1 = (iy) % am.ny + 1;
  k2 = (iy + 1 - k1) / am.ny + 1;
  if (value != 0 && g >= am.glow - 1 && g <= am.gup - 1)
```

```
      if (value > 0 && exteOpt(ie + 1, ies)
          &&
        ib == ibs - 1 && ((sys == 1 || sys == k2 + 1)
           &&
        k1 == 2)) {
      c2[sd] += value;
      }
}
```

The `gotValue()` function is executed automatically by the parent when a value is found. It gets the value and a stack containing the indexes. The first thing to do is to recover the indexes from the stack. These are the indexes of the sparse matrix for this value, usually used to add conditions or as sources for calculations. Then, we start the actual processing, which, in this case, is just accumulating this value in the output array `c2[]` if some conditions are met. The function `exteOpt()` comes from the *el teu* and is implemented in the `AmigaService` as well.

2.2.5 Statistics of the observatory component

The model component consists, at the moment, of 19354 lines of code distributed in 352 classes organized into 35 packages. These classes have 634 attributes and 1350 methods. There are also 1574 lines of code in 41 `UiBidner` XML files defining views and widgets.

Chapter 3

Performance

Software profilers are very useful tools to detect and fix performance problems in computer programs that lead to memory overflows or poor response times, usually caused by processing bottlenecks and memory leaks. These tools are usually run at the end of the development stage and set a starting point to the successive post-development iterations that will ultimately lead to the final product. Even though the system is still under active development, as are the AMIGA and the *look*, we considered it a good exercise to apply some profiling to the current state of the application in order to picture the overall behaviour of the system in a couple of fields, and tackle reported problems, if any.

We used jProfiler[1] to capture memory behaviour and file operations in a simulated real-life use of the AMIGA High-z Observatory. We did not profile CPU, for we consider the nature of this application does not make it CPU-intensive, but it rather needs to handle data and files in an efficient, rapid way.

3.1 Memory performance

In this section we intend to prove that the memory consumption stabilizes after a certain time of browsing the application thanks to the methods described in section 2.1. These methods, as we explained, perform the fetching and caching of data from files into memory so that the loading of data only takes place once in two nearly consecutive requests. Also, it releases memory resources if the cache size grows over a certain size.

To do so, we shall plot memory telemetry graphs, which display the amount of memory allocated and used in either the heap and the non-heap spaces. The heap space can be seen as the regular memory space of an application that holds data, object instances and other application-defined resources.

[1] Java profiling tool developed by ej-technologies (http://www.ej-technologies.com/). We used an evaluation version of ten days that was later kindly extended to ten more days under my request.

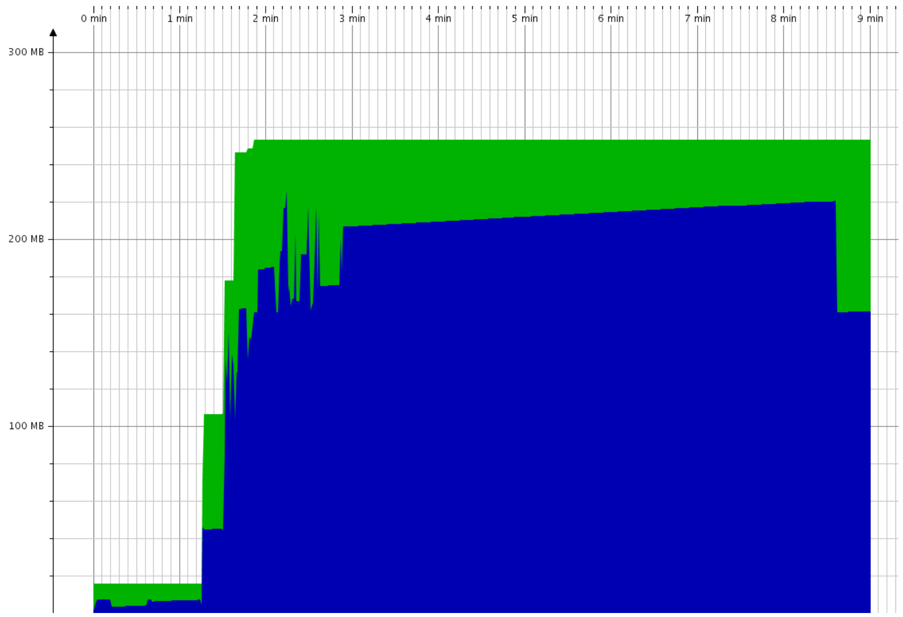


Figure 3.1: Memory telemetry plot of the heap space. 10 minute sample. The green area represents the amount of memory allocated by the JVM and the blue area represents the memory actually being used by the application.

In the plot, the blue space is used memory actually filled by data, and the green space is the memory allocated by the JVM[2]. The blue space experiments a sudden, steep increment in the beginning, when the cache is empty and thus the system must actually load data into memory, but then this increase stabilizes, given that successive requests find most of the data in the cache. The decrement at the end responds to the execution of the garbage collector, the GC, which sweeps out unreferenced objects and other unused resources, freeing memory.

If we take a look at the non-heap space, we see a similar behaviour. The non-heap space is used to store per-class structures such as runtime constant pool, field and method data, and the code for methods and constructors, as well as interned Strings.

[2] Java Virtual Machine, runtime environment into which Java programs run.

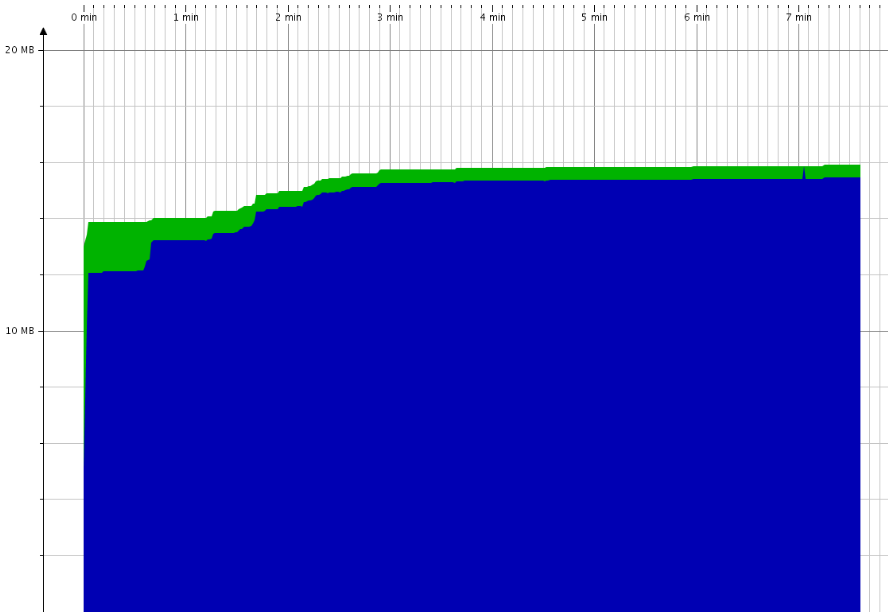


Figure 3.2: Memory telemetry plot of the non-heap space. 10 minute sample. The green area represents the amount of memory allocated by the JVM and the blue area represents the memory actually being used by the application.

In this case most of the actual loading happens at the beginning, for most classes and resources are loaded when the program starts. The smooth curve responds to the allocation of service item classes and other minor issues. The scale in both plots is not the same. While the non-heap space barely reaches 15 megabytes, the heap space extends well over the 200 megabytes.

3.2 I/O performance

In order to inspect the performance of I/O we need to monitor both the read/write operations on files and the throughput, which measures the rate of read bytes per unit of time. The run scenario is exactly the same as in the previous case, in fact I captured all the data in a single run.

The following table contains the actual access time line for each file that is used during our test.

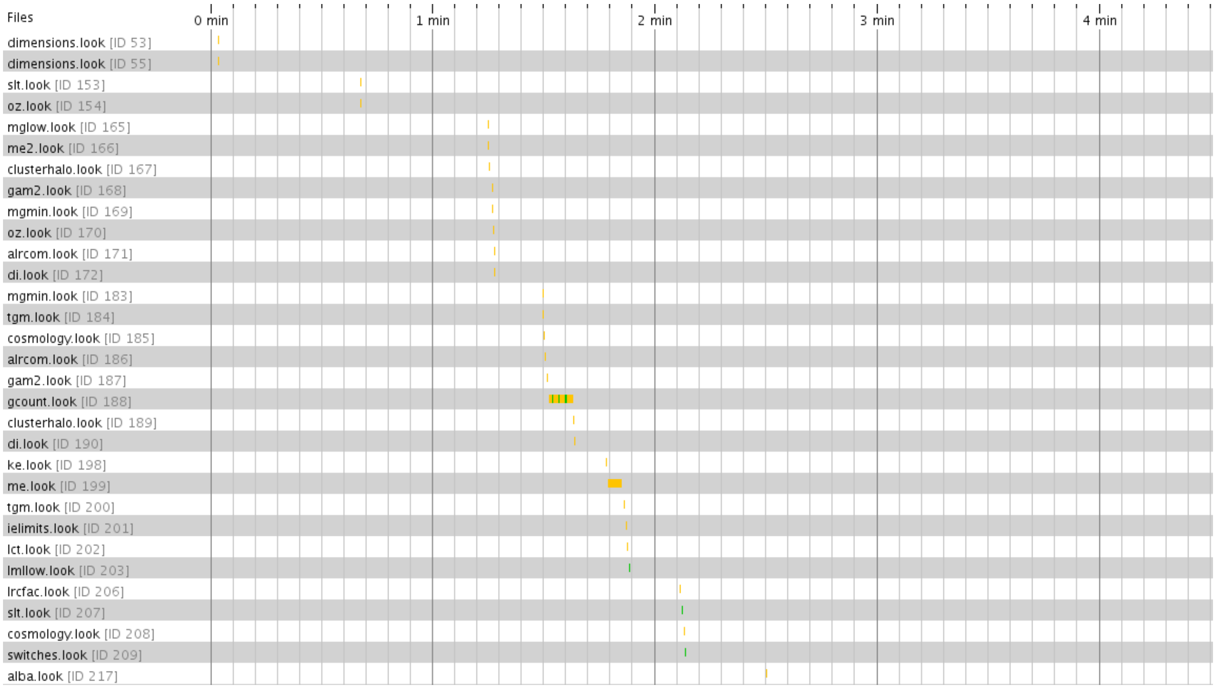


Figure 3.3: Time line of open/read/close operations on files. Yellow lines are *read* operations and green lines are *open* operations.

Obviously, larger files such as the `gcount.look` ($\sim 87MB$) or the `mr.look` ($\sim 6MB$) take longer to load (about 10 and 5 seconds respectively), but they are only loaded once per set and then stored in the cache. We can appreciate that after two and a half minutes of intensive use and item browsing, no more files are loaded, for most of them are already in memory. Not all files have been loaded at this point, but the number of disk access operations decreases dramatically with time thanks to the cache.

If we look at the throughput, we can see a big peak corresponding to the loading of `gcount.look` followed by a smaller one produced by the `me.look`.

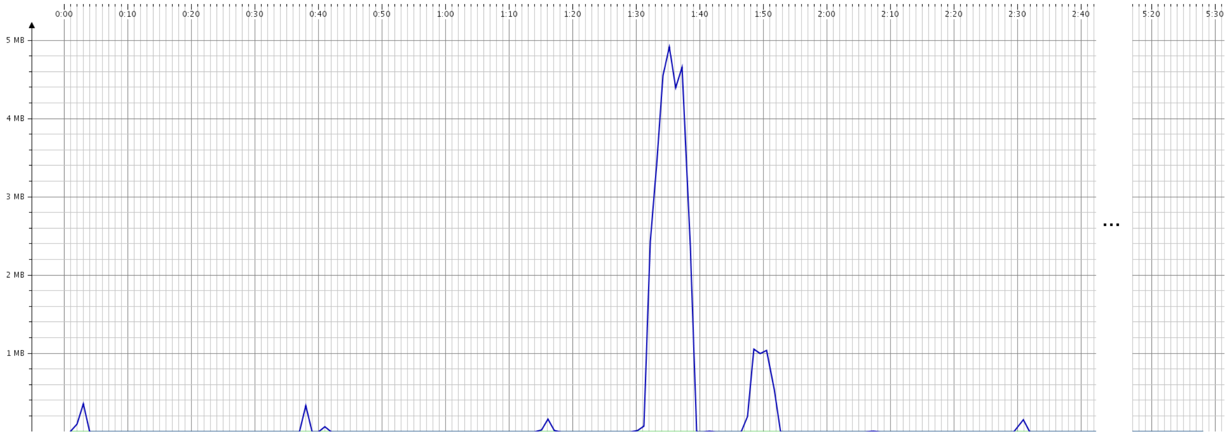


Figure 3.4: Throughput achieved for file operations in versus execution timeline.

Again, thanks to our file handling and processing methods, the disk throughput stays within reasonable limits, which is good considering disk access is among the slowest operations a computing system can perform.

Chapter 4

Conclusions

The AMIGA High-z Observatory is designed with an eye to the future, for its modular structure make it very easy to modify the data, change portions of the view or alter the way results are displayed. In a few words, the design of the application ensures its scalability and maintainability. Additionally, the system performs acceptably and has few memory and computing requirements -actually it can run in any mid-range desktop computer, which was one of the initial requirements- and offers a neat and usable interface. With this project we have achieved the purpose of giving a public facade to the cosmological semi-analytic model AMIGA. It is to be, with high probability, the first of its kind: the first to offer access to a general-purpose cosmological model and the first to be publicly available in the web.

Bibliography

[CL02] Peter Coles and Francesco Lucchin. *Cosmology - The Origin and Evolution of Cosmic Structure*. Wiley, 2002.

[EVO] EVO. European virtual observatory. http://www.euro-vo.org/.

[FBBO] Pierre Fernique, Thomas Boch, François Bonnarel, and Anaïs Oberto. Aladin sky atlas. http://aladin.u-strasbg.fr/java/nph-aladin.pl.

[Goo] Google. Google web toolkit. http://code.google.com/webtoolkit/.

[HNS$^+$02] M. Hirschmann, T. Naab, R. Somerville, A. Burkert, and L. Oser. Galaxy formation in semi-analytic models and cosmological hydrodynamic zoom simulations. *MNRAS*, 000:1–23, 2002.

[ISO06] ISO. Ergonomic requirements for office work with visual display terminals (vdts). http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=16883, 2006.

[Lau07] Ole Laursen. flot plotting library for jquery. http://code.google.com/p/flot/, 2007.

[Leo09] Alexander De Leon. Gwt adaptation of flot. http://code.google.com/p/gflot/, 2009.

[Lid03] Andrew Liddle. *An Introduction to Modern Cosmology*. Wiley, 2003.

[NVO] NVO. Us national virtual observatory. http://www.us-vo.org/.

[SBNM] Alan P. Smale, Roger Brissenden, Phil Newman, and Laura McDonald. Skyview, the internet's virtual telescope. http://skyview.gsfc.nasa.gov/cgi-bin/query.pl.

[VAO] VAO. Vao data discovery: Datascope. http://heasarc.gsfc.nasa.gov/cgi-bin/vo/datascope/init.pl.

[Wie72] Steven Wienberg. *Gravitation and Cosmology*. Wiley, 1972.

[WR78] S.D.M. White and M.J. Rees. Core condensation in heavy halos - a two-stage theory for galaxy formation and clustering. *MNRAS*, 183:341–358, May 1978.