Astronomical Data Analysis Software and Systems XII
ASP Conference Series, Vol. 295, 2003
H. E. Payne, R. I. Jedrzejewski, and R. N. Hook, eds.

Optimizing the Performance of ISO and XMM-Newton Data Archives

José Hernández, Christophe Arviset, John Dowson, Pedro Osuna and Aurèle Venet

ESA, Research and Scientific Support Department, Science Operations and Data Systems Division, Villafranca del Castillo, P.O. Box 50727, 28080 Madrid, Spain

Abstract. In this paper we try to give a quick overview of the techniques used in the development of the ISO and XMM-Newton Data Archives in order to optimize their performance.

1. IDA and XSA Quick Overview

1.1. ISO Data Archive

The ISO Data Archive[1] (IDA) has been developed at the European Space Agency ISO Data Centre at Villafranca, Spain. We believe that this archive has proven valid a new concept in the world of astronomical archives. Instead of the more commonly used HTML/cgi-bin based approach, the IDA was developed based on a "three-tiered architecture" (see Arviset 2003) and using the Java language on the Client and Middle tiers. This choice was not obvious in 1997 when the development of the archive started, however it has been very successful, according to the feedback received from archive users.

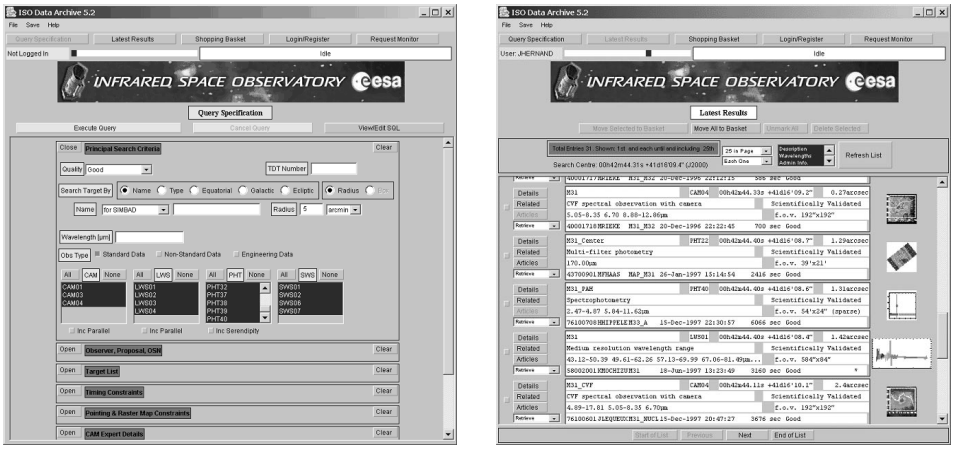

Figure 1. ISO Data Archive.

[1] `http://www.iso.vilspa.esa.es/ida`

The IDA main characteristics can be summarized as:

- Developed using Java, certified 100 % Pure Java.
- The first version of IDA was released on December 1998, current version is 5.2.
- The archive has been extensively used by the astronomical community:
  - Over 1300 registered users.
  - Archive data delivered 3 times to the external world.
  - Over 100.000 queries performed from outside.
  - Accessed from all sorts of platforms: Solaris, all Windows, Linux, Macintosh, Digital Unix, OSF, HP Unix, IRIX, AIX,...

1.2. XMM-Newton Science Archive

The XMM-Newton Science Archive[2] (XSA) has been developed at the European Space Agency XMM-Newton Science Operations Centre at Villafranca, Spain. Thanks to the choices made in the development of the ISO Data Archive, it was fairly easy to reuse its architecture and a great deal of its code. This led to a very short development time (less than a year), a more functionally rich system and a product that can now be more easily reused for other Data Archives.

The XSA archive has over 500 registered users. Some of the improvements made in its development were also applied to the IDA archive. Having a similar look and feel in the archives benefits both the end user and ESA. The archive user does not need to learn how to use a new interface and for ESA it helps in strengthening its corporate image among the scientific community.

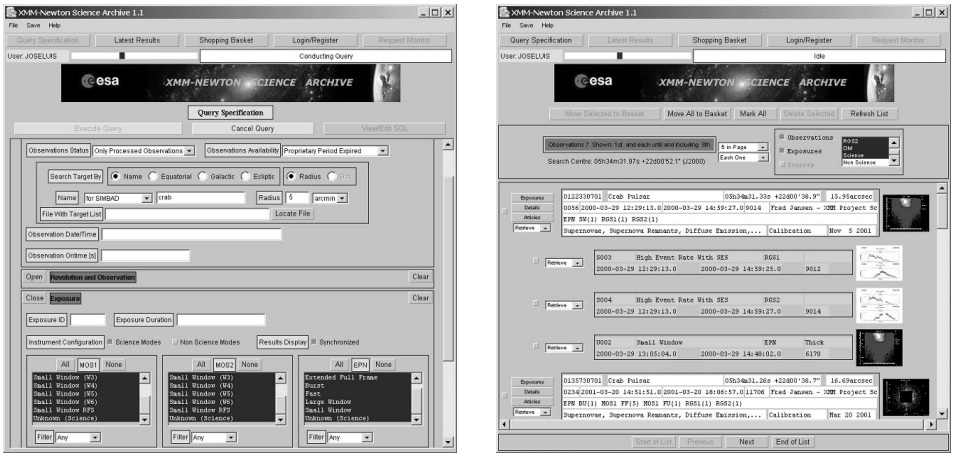

Figure 2. XMM-Newton Science Archive.

[2] `http://xmm.vilspa.esa.es/xsa`

2. Performance Optimization Techniques Used

2.1. Optimizing the Applet Size

The ISO and XMM-Newton archives are accessed by the astronomers using a Java applet that gets downloaded to the user's computer and runs locally inside his browser. By using Java, one has access to powerful graphical capabilities and a very configurable and user friendly interface can be provided. However, Java applets can be very large in size leading to a bad user experience as he has to wait until the download is completed before starting the application. In order to minimize the size of the applet we have applied several techniques:

- **Build a thin client**. The client only takes care of gathering user input and presenting results. This, apart from other advantages, reduces the number of classes needed on the Client Tier.
- **Use obfuscation tools**. These tools reduce the size of the classes as they generate shorter symbol names. We used a commercial tool called Jshrink[3] and typically we get reductions in size of approximately 15%.
- **Use more than one jar file**. Only one is needed to start the application:
  - Expert functionality is downloaded on demand or in the background.
  - We use three jar/cab files of sizes 250, 75 and 175 Kbytes.
- **Use the standard Java 1.1 API** as much as possible. Don't write new classes when it can be done using the API.
- Even if the applet is Java 1.1 compliant, **compile it using Java.1.2** compilers as they generate more efficient byte code (smaller & faster).

2.2. Optimizing the Applet Execution

The portability of Java comes from the fact that the Java compilers do not generate machine dependent code but byte code that gets interpreted at runtime by the different Java Virtual Machines. This means that some performance is sacrificed for the sake of portability. However, the Java interpreters have become more and more efficient and the difference between the execution of Java byte code and machine code has narrowed.

In any case when one deals with applications distributed across the Internet, like our archives, the bottleneck is usually the network. Thus, great improvements can be expected when efforts are concentrated in improving the efficiency with which information is moved between the application layers.

These are the main techniques we have used in order to boost the application execution:

- Concerning the data transfer:
  - **Minimize the number of client-server interactions** by grouping them. Due to the overhead involved in creating a new connection it is more efficient to have a few big data interchanges than a lot of small ones.
  - **Persist client session information on the Middle Tier** to avoid unnecessary data interchanges.

[3] `http://www.jshrink.com/`

- **Embed initialization information in the client** jar files.
- **Compress all the data transferred** between the client and the server tiers, the standard Java API comes with classes that make the compression easy both on the client and the server. We are compressing all the serialized objects and obtaining reductions in size by a factor of four.
- Use the **transient modifier inside the Java classes** to avoid writing unnecessary data into the streams. Sometimes, overwriting the default serialization methods also helps.

- Concerning the application execution:
  - **Reduce the number of classes used to start up the application**. Then load the rest of the classes in the background or on demand if the user asks for extra functionality.
  - **Use Lightweight graphical components** instead of native ones. Note that this point conflicts one of the points mentioned before about using the standard Java API classes.
  - **Use the time the application is idle** to build the structures that will be needed later. For example when the application is started and before the user performs a search to the archive, we build the GUI elements that will be used to display the search results.
  - **Reduce the number of queries performed to the database** by grouping them and use cached statements when possible.
  - On the server tiers **use the latest Java Virtual Machines with Hotspot** turned on to improve the server performance.

2.3. Conclusion

We believe that Java applets and applications can show a very good performance when a few rules are followed, it is advisable to consider performance issues early in the application design. Most of the performance benefits come from considering the network as the bottleneck.

Trying to optimize *a posteriori* by rewriting bits of the code is not a good idea, usually one ends up complicating the application code and introducing bugs that are difficult to spot and fix. Before tackling this sort of optimization it is better to make sure that it will produce a significant improvement.

References

Arviset, C. 2003, this volume, 47