Atomsnet Developer Manual

Willem de Bruijn
Leiden University

July 2002

Contents

1 Developer Manual

The atomsnet application has been developed as an open source project. All code is made freely available under the GNU General Public License. As such, I want to encourage anyone to learn from, alter and reuse any code. To allow program alterations the application has been designed as modularized as possible. How these modules interact is discussed in detail in later sections.

I've tried to write useful comments in between the sourcecode wherever applicable. This should allow anyone with a basic knowledge of programming to start coding right away. Therefore this document will only describe the architecture from a bird's eye perspective. Detailed information about the application's functionality can be found in the User's Manual. This information will not be repeated here, so please read that document before continuing.

Environment Atomsnet has been written as a Windows application using standard C++. I've used the Microsoft Visual C++ 6 integrated development environment for creating the system. Apart from the standard Windows hooks the system also depends on the Microsoft XML parser, version 3 or higher. Information about all things microsoft can be found at their developer's website, http://msdn.microsoft.com/ .

Porting Reusing code in another environment has been made relatively easy by shielding most operating system specific calls from the general code. All library loading, networking and window manager specific calls have been placed in specialized functions that can be overridden with limited effort. The XML parser calls exist solely in the datalibrary module. Unfortunately they can be found throughout this module, so it will be harder to replace them.

1.1 Plug-in Architecture

Metadata indexing is based largely on file-type specific heuristics. Therefore I've chosen to remove all indexing code from the application codebase. The actual process of finding metadata has been placed in separate modules that follow a very basic design. These modules shall from now on be called plug-ins.

The application's plug-in architecture consists of two parts: the module loader framework and a selection of representative example modules.

1.1.1 Framework

The framework is responsible for:

finding modules
verifying module integrity
loading modules
running modules
integrating module results in the database

and
safely unloading modules

The functionality is encapsulated in two functions. Upon opening a database, the framework is loaded for the first time and the discovery loop is entered. This loop reads all files in the plug-in subdirectory of the homedirectory. Then it tries to load each file and verifies its integrity by calling a standard function. This somewhat rudimentary testing should suffice to accept or deny any file encountered. More importantly, crashing the system by loading incorrect files should not be possible. Each file that has been identified as atomsnet plug-in is added to the opened database under a special key.

When indexed, files are normally added to the document as xml elements called resources. These elements are placed in the tree-like structure according to their metadata. Plug-ins are added in largely the same way, but are distinguished by their element's name. This mixing of data and code will be discussed in the next paragraph. After initialization all plug-ins are released and unloaded from the execution environment.

When a user drops a file the actual indexing takes place. This is where the plug-in/resource mixing of the datatree becomes useful. Upon starting the indexing process no previous data exists so a general plug-in is called that resides at the root of the datatree. This plug-in is guaranteed to return a result, since it relies solely on filesystem information. After this bootstrap process a recursive process starts where the framework searches through the tree for an applicable plug-in. The search is started from the location where a resource has last been added or altered and continues up the tree. This way all ancestor nodes are travelled. When a node is encountered containing a plug-in this plug-in is called for the current file.

A problem that may occur with this recursive process is that a loop in the dataflow can be created. This problem has not been dealt with explicitly in the testprogram, since too few plug-ins currently exist to create such a loop. Please note that it should ofcourse be dealt with in reallife products where more plug-ins are used.

The rationale behind the recursive indexing process is that recently discovered information can help us select new appropriate plug-ins. This is obviously at least as efficient as calling all plug-ins, while it gives at least as good results as calling a preselected subset.

Let me give a small example. A user wants to index an mp3 file. In the standard package the following process would take place. The bootstrap plug-in writes filesystem data and then calls the MIME-type plug-in, because it is entered at the root of the filesystem subtree. The filesystem plug-in is not called again since recursive plug-in searching stops one level above this node (a somewhat dirty hack). The mime-type plug-in enters information under the audio/mpeg entry of the file-type subtree from where the mp3 specific plug-in is called. Finally, the google plug-in is called because it is entered at the root of the MIME-type subtree. It is obvious that new subtrees can thus be added to the system by creating an appropriate plug-in. Similarly, subtrees can be removed by removing specific plug-ins. This flexibility is the main strong-point of the plug-in based system. The next section deals with coding a plug-in.

1.1.2 module layout

The framework uses a very simple scheme in communicating with the plug-ins. The rules have to be followed precisely, otherwise the system might either not accept a plug-in or show unexpected behaviour. The precise naming of the functions can be seen in the example plug-ins source. I will now only describe the general functionality required. It is probably easier to alter the example plug-ins then to create a new plug-in from scratch. There exist two different versions, however. These two versions have identical hooks when it comes to displaying their name, version and preferred locations. The difference lies in the actual processing hook. The more advanced plug-ins adds XML data using the MS XML API. Since plug-in developers might not want to use the MS XML parser a plug-in is also accepted if it listens to a second function, but not to the first. This second functions is expected to return a simple text string containing the xml data.

The framework starts by calling the simple text function. If this function returns an empty string or the NULL pointer, the more advanced XML enabled function is called.

It is completely up to the developer whether he wants to create XML enabled plug-ins. The main advantage for the developer is that he can view and alter the open database from within the plug-in and thus has more options than with the the text based plug-in. If this extra functionality is not needed I would suggest creating pure C++ text based plug-ins, since these are quicker to write, smaller in code and easier to port.

1.1.3 example plug-ins

Since this is a testprogram only a few plug-ins have been created. The most simple one reads data from the filesystem and creates an entry in a subtree that models the filesystem. The second plug-in uses file content to identify the filetype and creates an entry in a MIME based hierarchy. This module is currently a simple wrapper around a function exported by the *Microsoft Internet Explorer*^{tm}. It identifies some 30 different types. A third plug-in is created specifically for reading the proprietary metadata format often used in .mp3 audio files called ID3. This plug-in does not add items to the hierarchy, but simply alters and extends existing entries.

The last plug-in best demonstrates the strength of the system. This plug-in creates keywords from the already added information and with these keywords queries an internet directory. The directory queried is currently the google directory (directory.google.com) that is equivalent to the open directory project. From the answer a local version of the directory is created containing the processed files.

1.2 Reusing Modules

Besides the fine-grained plug-in architecture, the application itself has also been subdivised into larger modules. All user interface code can be found in the executable, while the database and networking code has been placed in separate libraries. These are called datalibrary.dll and netlibrary.dll .

Modularizing the entire application allows for relatively easy replacement of the interface, the networking approach or the database back-end. It also helps in keeping the interdependency of code relatively small. The data- and netlibrary are called by the user-interface through special hooks to start and stop their specific routines, such as the webserver and the indexing engine. Furthermore, the database library contains a class that is used by the webserver to query the database.

Which functions have to be exported can be seen in the .def files accompanying the module sourcecode. All interfacing is done through standard C++ function calls, so portability should not be to great an issue.

1.3 Database layout

The databases used by the atomsnet applications are standard XML documents. This reduces system complexity, since data can be sent to a client without conversion. Formatting this data into webpages is done on the client's computer through the use of XSLt transformations. More about this client side processing can be found in the User's Manual.

The database layout follows a strict grammar, which has been written down as an XML schema document. Due to the poor validation engine currently part of the XML parser this document is not yet used by the application. Developers are, however, expected to comply to this guideline, so please read it before changing the database layout. The schema document can be found at http://atoms.sourceforge.net/atomsnet.xsd .

To summarize the grammar, the document consists of a standard namespace header and an ¡ rdf ¿ .. ¡ /rdf ¿ pair encapsulating the actual data. No actual rdf validation is used, but adherence to the Resource Description Format is encouraged. Within these tags the ¡ atomsnet ¿ .. ¡ /atomsnet ¿ tags set apart the atomsnet data from possibly other rdf data. This is the root of our actual database. It consist of an application- and a datatree. The application part contains accounting information, such as the number of nodes in the database and the server's ip address(es). The database node can have a possibly unlimited number of subtrees, but contains only 3 in the base distribution, namely the mime, cat and sys trees. These contain MIME-type, Open Directory Project categorization and filesystem dependent information, respectedly.

Each subtree of the database can contain a possibly unlimited number of ¡ resource/ ¿ nodes, describing files. They can also contain a possibly unlimited number of plug-ins as described above.

The resource nodes can contain both subnodes and attributes describing metadata. In the reference version of the application all data is written in attributes. One of these attributes, the ID, works as a key for quick lookups. It is especially useful when finding different resource tags belonging to the same file. As an example, the ID s1 responds to the file with ID 1 in the sys subtree. A resource with ID m1 will have to refer to the same file as the s1 resource, but will reside in the MIME-type subtree.

Apart from the features described in the schema the database can be extended and altered to your personal need. The restrictions laid upon the grammar are needed to allow interoperability between atomsnet applications network wide. An example database containing one file is shown below.

```
<?xml version="1.0"?>
<?xml-stylesheet type='text/xsl' href='ane_html.xsl'?>
<rdf>
<atomsnet ip="http://0.0.0.0:80">
<programdata version="1.1" idcount="1"/>
<index>
<sub name="mime">
<plugin name="anedll_cat_dir_google.dll"/>
<sub name="audio">
<sub name="mpeg">
<plugin name="anedll_mimempegaudio.dll"/>
</sub>
</sub>
<sub name="application">
<sub name="octet-stream">
<resource id="m1" sysref="s1" catref="c1" name="atomsnet v1.lnk" size="378" date="2002-7-7" artist="atomsnet v1"/>
</sub>
</sub>
</sub>
<sub name="sys">
<plugin name="anedll_mime.dll"/>
<sub name="c:">
<sub name="windows">
<sub name="start menu">
<sub name="programs">
<sub name="atomsnet">
<resource id="s1" mimeref="m1" name="atomsnet v1.lnk"/>
</sub>
</sub>
</sub>
</sub>
</sub>
</sub>
<sub name="cat">
<sub name="computers">
<sub name="software">
```

```
<sub name="operating_systems">
<sub name="network">
<sub name="distributed">
<resource id="c1" mimeref="m1" sysref="s1" artist="atomsnet v1"/>
</sub>
</sub>
</sub>
</sub>
</sub>
</sub>
<plugin name="anedll_sys.dll"/>
</index>
</atomsnet>
</rdf>
```

1.4 Further information

Apart from this document there exist a user manual and a research document based on the application. Furthermore, the application is also being made available as sourcecode, meaning that you can view precisely how it works. For this an understanding of C++, Windows API's and webstandards is very useful.

If you have specific questions you can also ask them directly by placing a message in the discussion board at the atomsnet website. The address is http://atoms.sourceforge.net/ . Remarks and bugreports can also be placed there.