DSpace Statistics eporting Design Thoughts

There are three different types of statistics that we would like to address:

- Archive Statistics - eport on the contents of the archive
- Access Statistics - eport on archive usage
- Administrative Statistics - throughput of work and time taken, etc.

Currently log4j records the access statistics in the following form:

```xml
<timestamp> <log level> <logging class> <user> <session info> <log string>
```

where `<log string>` is typically split into the form:

```xml
<action><parameters>
```

Both of these are defined by the programmer at some point in the code when the log line is generated.

Should we have rules as to when logs are generated, and which parameters are passed. What would the criteria be?

Two likely choices

1. To abandon the log4j logging framework altogether, and write a full logging system for DSpace
2. To periodically load the data from the log4j logs into the database for analysis

For (1) the pros and cons seem to be:

**Pros:**
- We get exactly what we want out of the logging system
- We will be able to generate up to date reports on the fly in a customisable environment

**Cons:**
- A lot of work is required to build such a system, including threading of log writing, import and export of logs from the old system.

For (2) the pros and cons seem to be:

**Pros:**
- Not a major front end performance hit at any point
- No need to write a new logging system

**Cons:**
- No instant up to date statistics reporting
- We may not get all the information that we would like

The basic flow of functionality for each of these options would be:

(1):

```xml
+.....................................++-------+ +-------------+ 
```

(2):

```xml
+----------+ +--------+ +-------------+ +----------------+ | DSlogger |---->| Log D |---->| LogExporter |---->| dspace.log.xml
```

The logging database

Either way, it would be best to hold the data from the logs in a database to allow for dynamic and complex report requests. This is a proposed database structure for holding log information:

```sql
logs----log_id int auto increment PK -- the id of the log lineip text -- the ip address associated with the requesttime
timestamp -- the timestamp of the request timeaction text -- the action of the requestrobot boolean -- whether the ip is a known robot or
notlog_params----------param_id int PK -- the id of the parameterparam text -- the human readable parameter namelog_param_values----------------
param_id int FK -- the id of the parameter that we are attaching a value tolog_id int FK -- the id of the log line that we are attaching a value
tovalue text -- the value of the parameter
```

This is a fairly basic triplestore method, and aims to allow multiplicity of parameters per single log line. It also allows for a registry of all passable parameters.

Some questions regarding this structure and associated considerations:

- How would this cope with exception handling? Would we want a separate log table to deal with stack traces, and if so what should it record?
- Perhaps the stack trace and the environment in which is was generated (from the HTTP header), for debugging purposes?
- Do we want to record robot IPs at all? Should we pre-filter these, then we don't need the robot field in the logs table? The downside is that this means the logs are not complete, and disregard the situation where you might want to know how many times and in what ways your archive has been crawled.
- Where can we get a list of robots from? Do we need to maintain/hold our own, or are there services for this?
- How will we manage the parameter list? I have some half thought out methods explained shortly.

Possible solution to log parameter handling
It would be relatively straightforward, at this stage, to define a list of the most likely parameters that will be passed to the logging system, since much of the DSpace interactions happen between the already existing system elements. Therefore, one possible solution is to define a Log Params class which knows what the core system parameters are. As a basic stab at the default list, this might contain references to the following log parameters:

- user
- collection
- community
- group
- bitstream
- item
- bundle
- order (as in sort_order)
- result
- handle
- query
- results

Each of these would probably be an integer reference to the database id of the system object. A question arises over the persistence of some of these objects, though, especially users and groups.

A possible alternative to this is to allow each module or class to define an associated Module Params class which knows which parameters (and actions) the module will be able to pass to the logging system. At module installation time, these parameters could be written to the parameter register via a Log Params class. We may then even wish to go so far as to define log writing facilities in the Module Params class, such that you could call:

```java
Unknown macro: {ModuleParams.myAction("param1", "param2", ...);}]
```

So for example:

```java
Unknown macro: {DSLogger dsl = new DSLogger("myClass");dls.log(LogParams.WAN, ModuleParams.editItem(item.getID()));}]
```

The problem with this is that it seems overly complicated, but it will at least enforce good logging and require developers to think about what their logging requirements are. Hopefully this will enhance the logs rather than causing people not to use them.

**XML based log file**

Assuming that we have a new logging system for DSpace, it will be necessary to generate logs in files at some point for the purposes of being sure they are backed up and are transportable between systems/locations. In addition, it has been suggested that using XML to hold log file information could be a good way of aggregating logs from other sources, allowing for usage of particular system elements (especially items) to be monitored across a number of DSpace instances which may share data.

Perhaps the log file could look like this (although it is relatively long winded in comparison to the log4j files):

```xml
<logs start="start date" end="end date"><log time="timestamp" ip="ip address"><action name="action"><param name="param1" value="value1" /><param name="param2" value="value2" /></action></log>...</logs>
```

**Administration statistics**

Generating statistics for administration may require a slightly more complex data model. To allow a degree of flexibility in how the data is stored we could use a similar triplestore method to the one used above for parameters. Thus the database structure may be:

```sql
admin_stats-------------stat_id int auto increment PKitem_id int FKadmin_properties----------------property_id int auto increment PKproperty textadmin_values------------stat_id int FKproperty_id int FKvalue text
```

The only systems I can think of straight off which require this sort of statistical analysis would be the Submission process and the Work Flow. For each of these you might want to know really basic information like how long it took to pass through this process, and which users were involved. A really basic way of doing this would be to store data such as the start point and the end point and the user ids of any users involved. Therefore your properties would be something like:

- start
- end
- user

If we want to do more complex things like determine what state the item was in for what time periods and under the jurisdiction of who, then the data model will need to be re-thought.

**Report Generating and other UI considerations**

Basic report generating could be done with a set of stored queries, or the option to input new queries, or a straightforward query interface. It may also be desirable to produce executive summaries and other sorts of reports with nice standardised forms for the output. This could be done by defining a layout template for result sets and pushing them through that before going to the main DSpace UI. This section would need to be fully i18n compatible, so it would probably be necessary to have entries in a Messages properties file which map actions (and parameters?) into specific languages. It may also be sensible to generate common reports periodically anyway, so that the flat HTML files can be served directly with no other impact on the system. A nice consistent statistics navigation would also be useful, to move through stacks of pre-generated reports or pre-defined queries.

</html>