DSpace Architectural Futures: A White Paper

Robert Tansley, 2006-04-27

1 Introduction

This document is an attempt to get down on paper the various thoughts that have been bubbling around my head over the last 2-3 years on the DSpace architecture, with the intention that it can provide a useful set of pointers and guidance to the DSpace technical architecture committee in their deliberations. Time considerations mean that it is not necessarily a fully-baked document with references and so forth, and I just use technical terms (e.g. from OAIS) without a glossary, but should still prove a useful document.

I've a feeling this paper will end up containing rather more questions than answers; however this will probably more useful in terms of guiding future architecture work than an architecture proposal.

The thinking in this paper has in mind a broad scope for DSpace; while the focus thus far has been on "institutional repository" (as generally understood in the US, as opposed to the more open access-centric UK view), uses of DSpace has broadened considerably to areas such as museums, corporate use, and government organisations such as law libraries. At each stage, how much effort is expended to support each stakeholder should probably depend on how much that stakeholder itself can invest; it's probably not viable, for example, for government organisation use cases to be supported if all resources for development and maintenance come from the academic library community. However, for architectural visions and directions, to focus too much on one silo of use and to exclude these other use cases would limit DSpace's potential impact. The governance advisory board produced a mission statement that extends to this broader mission. The vision should be grand and broad.

Further, the kinds of data that are being stored and could potentially be stored by DSpace are limitless. It really is an effort looking at how to manage and preserve the new ways that people create and communicate. Don't just think of sets of PDFs; what about videos, satellite data, interactive objects, software? We will need to work out how to manage and preserve these things (even though there is a tendency to be over-paranoid about losing JPEG specs forever, or what a horrible loss it is if the photo appears two pixels to the left of where it should in a future rendering).

All this said, one should not try and boil the ocean. There are always, always, always *trade-offs* between flexibility, usefulness and simplicity. Refusal to accept limitations and trade-offs is the fastest way to ensure that you end up with a bloated, impenetrable mess of a system. This leads to the single biggest challenge that the DSpace community faces moving forward:

2 The Balancing Act

As DSpace's usage has increased, it's not surprising that people have run into limitations, accompanied by numerous questions as to why it doesn't do versioning,

locking, support all metadata under the sun (whatever "support" means) and so forth. While there are some aspects that can clearly be put out of scope, with most it's a big grey area, and in many areas, there is no established best practice or mature knowledge about how to accomplish something. There are various dimensions that DSpace needs to be carefully steered along.

2.1 Simple versus complex

Of course the most challenging is whether to make a simple, understandable system or a complex, powerful system that has everyone scratching their heads. Throughout my 5 1/2 years on DSpace I've constantly pushed to keep things as simple, consistent (and well documented) as possible, to lower the barriers to use and development.

There are different dimensions to simplicity and complexity.

One dimension is in terms of ease of management and use. Regardless of code complexity, is it possible for a non-expert to bring up a DSpace and manage some content in it over time? If they have to come up with their own data models (or choose from a bewildering array), maintain a fine-grained, complex format registry, a non-expert will soon get confused. Maybe such non-expert users aren't in DSpace's target audience, and we really should be trying to make DSpace a heavyweight digital curation tool that needs experts to manage – however, I think non-expert users form a key part of the DSpace community right now and so the barriers should be kept low (but the possibilities extended within this limitation).

In terms of technical architecture and code, complexity can be hidden, abstracted, layered, "modularised" etc. etc. etc. but there are always trade-offs. For this open source effort to succeed, developers need to be able to look inside DSpace and understand what is going on, and the more complex it is, the harder this will be. Likewise, for users, if DSpace is too complex to bring up, or figure out how to get your stuff in and out of it, adoption and therefore investment will suffer.

Being too simple is no good either! It's identifying the places where you don't need to be complex that's the trick, and some well-meaning over-design can cause real problems. Don't be afraid to throw away old unneeded code.

2.2 Experimental versus proven

Even now some aspects of DSpace are rather more "experimental" than others, for example the History system. Although DSpace is a pretty robust system right now, it won't be able to fulfil its mission, and will not exploit this open source research to deployment channel I've droned on about, if there is not room for research and experimentation. We should be careful to note which areas are more experimental than others. Of course "make it modular" (easier said than done) will help here but the project will have to make various fundamental calls at some time or other and explore directions for the very core of DSpace so the issue won't disappear.

2.3 Middleware versus Application

Right now DSpace is pretty firmly in the application domain – certainly not middleware like Apache HTTPD, an application server or relational database. To support the various emerging use cases, it seems clear that there needs to be a 'core' of functionality, with a periphery of application-specific components. The question

becomes which parts of this whole picture are "DSpace"? Which parts are the responsibilities of the structured open source community to maintain? How can the critical "out of the box" functionality be maintained?

Back in March 2004 I talked about the idea different DSpace "distributions", with a common core but with different configurations and installed/components. DSpace would essentially become something more like GNOME than a single app, where a set of core functionalities is built on by other components which provide the user functionality. I still think this has merit, though it may take a while before we have the maintenance bandwidth. In any case, the idea of which are components are "supported" by the DSpace (community/federation/foundation), and which are community contributed/maintained should be made clear.

2.4 Revolution versus Evolution

The ideas I laid out in March 2004 (which have largely been accepted with a couple of minor changes) were never meant to be a "new" system, though of course calling the new architecture it "DSpace 2.0" perhaps led people to think that way. It was never going to work for someone to hive off and create a brand new system, call it DSpace and expect the rest of the community to follow.

On the other hand, at the opposite end of the scale, changing a single line of code at a time is unlikely to yield the advances in the core that are required. It will be challenging to chart a course between large-scale refactoring and implementation while maintaining community confidence and contribution.

I am increasingly convinced that focussed, centralised development efforts will be required to move DSpace forward, but there should not be just one, and the efforts must be closely coordinated.

So, evolution, with mini-revolutions occurring in strategically targeted areas.

3 What's Right

Before delving any deeper into problems, issues and suggestions, it's worth pausing to reflect on what's *right* about the DSpace architecture. It couldn't have got this far without something reasonably robust under the hood.

First and foremost DSpace is an easily understood, easily installed, "out of the box" application. This has been critical to DSpace's adoption. Largely, anyone with a reasonable amount of system administration knowledge can bring up a functioning DSpace system within half a day or so, and start experimenting. People can see it, put stuff in it, play with it, and instantly have a handle on what it is and does. This low barrier to adoption has enabled its use in diverse settings where such technology might not previously have been considered, and established a foothold in that application space. If DSpace is to continue to grow into new areas, it is essential not to lose this property.

The data model underlying DSpace is simple, and pretty easily understood. This has again been a key factor in its adoption. More complex object models inevitably raise the barrier to successful adoption. Further, there is always a trade-off between *flexibility* of a data model on the one side, and how quickly people can understand it

and start using it, and how easy it is to manage, manipulate and build meaningful services on the other side. A model in which there are just arbitrary nested containers and files is flexible, yes; you can store any sort of super-complex object in that. However, it also leaves people scratching their heads as to how to put what they have in it. Further, it is very difficult to even present a meaningful UI over that stuff; end users probably won't be able to make much sense of a big tree of abstract entities and files. Approaches like providing standard templates can help here but the benefit that having an easily understood, albeit limited model that everyone can instantly understand and start chucking stuff into should not be understated.

The DSpace architecture is also cleanly defined – storage, business logic, interfaces that interact with users or other systems. As far as I can recall, there has *never* been a question on any of the mailing lists where people are confused about some aspect of it. (People may question aspects, or assert they can do better, but these are universal constants.) That said, there have been a few slips, for example where the storage layer is accessed directly from the interface/application layer, and these should be ironed out. (A few inconsistencies crept in during the early stages of DSpace open source development – we initially set the bar for contribution very low to encourage involvement.) Often people mention that there's too much business logic in the UI code – while this may be true, achieving a real separation of logic from UI is harder than most think.

In the preservation aspect of its mission, DSpace seems have been essentially successful thus far. This may of course be due to the fact that anyone who does lose content or have it somehow mangled is not likely to be apt to advertise the fact widely. However, significant losses in such public things as university repositories are unlikely to go unnoticed, and given the sheer number of deployments we can be reasonably sure that losses do not happen often.

Part of the reason for this is that it's pretty hard to inadvertently screw up a DSpace installation in a way that breaks the stored data. (Other than poking around with an SQL or shell command line, but people generally appreciate that that sort of thing should be done with immense care). It's fully transactional, even at the bitstream storage level – any amount of adding, deleting metadata or files can be rolled back, ensuring the system is always in a consistent state.

The "Context" object (from org.dspace.core) is central to this, although in retrospect, perhaps it could be better named. It stores user authentication info and the database connection, and all in-memory Java objects that represent an object in the DSpace instance are tired to a single Context object. When that Context object is terminated, those in-memory Java objects are no longer valid (as e.g. getXXXX() methods will no longer function as there is no longer a database connection). Some contributions have started to move away from this model, principally due to some dissatisfaction with the database access API, but this the Context object is and will remain a key concept in DSpace architecture.

One more aspect of note is that the entire internal state of a DSpace instance is held in the relational database and file system. (A minor exception is during an authentication operation, whereby the parameters from the original request that prompted authentication are stored, so that after successful authentication the request can be transparently carried out.) This property also helps make DSpace robust – if the server goes down, or is rebooted etc. DSpace springs right back up as if nothing

has happened. Also, this makes it easy to deploy DSpace on a cluster of servers fronted by a load balancer – each individual server doesn't have its own internal state.

I strongly suspect it may be necessary to change this particular aspect with a publish/subscribe kind of model, but this should only be done in such a way as to retain robustness and scalability in terms of clustering capability.

4 Areas of Responsibility

The DSpace community has a wide range of areas to cover, requiring a variety of expertise:

- Managing developer tools (SourceForge, Wiki etc)
- Technical support, for deploying (upgrading, managing customisations) and developers
- Educate and coordinate the community
- Develop end user functionality
- QA/bug fixing, code clean up
- Release management
- Documentation (for end users, curators/administrators, sysadmins, developers)
- Develop administrative/curatorial interfaces (key area and differentiator for DSpace)
- Develop the architecture to support richer metadata and content needs and support preservation
- Engineer an enterprise-scale application (robust, enabling clustering etc)

All of these areas need to be covered, and not all are receiving the attention they need. In particular, the last 3 bullets in the list are not receiving the attention and effort they deserve. These 3 are unlikely to be covered in an ad-hoc way; they will require sustained, dedicated, centralised efforts to make progress to a point where wider community participation can effectively take over.

5 The Data Model

While it's not *wrong* as such, the DSpace data model does need a lot of work. Some of the problems may be addressed by establishing best practices and processes and updating the code, rather than reflecting any inadequacy of the data model itself.

For example, people constantly bemoan the fact that DSpace doesn't "support" anything other than Dublin Core. That depends what you mean by support. You can add metadata in XML or RDF/XML or anything else as a bitstream. You can then add a Media Filter that crosswalks that to DC (or another flat metadata schema) and thus have it indexed. That's as much as most systems manage. However the problem is that there's no standard place to put the metadata. So documentation and established practices are needed.

In this section, I focus on the abstract data model, as opposed to its realisation in the relational database or METS. When I am talking about that realisation, I'll make it clear in the text.

5.1 Bitstreams

Not too hard a concept, though I wish we'd called them something else. ("File" would work just fine really.) But I think we're stuck with "bitstream" now.

What these things definitely need are better identifiers. The "persistent" URL is pretty fragile. An example bitstream ID is:

https://dspace.mit.edu/bitstream/1721.1/1564/1/foo.pdf

The key parts being of course the embedded Handle, and the sequence number inside the item. The other parts are necessary for systems to access the file and for browsers to do something sensible with the content.

This works fine for allowing other systems to directly access bitstreams, but is not robust to transferring between repositories, server changes etc.

The info: URI provides an alternative to enable globally unique IDs. Bitstreams can still be accessed via the above URL, but their main identifier should be something of this form:

info:dspace/1721.1/1564/bitstream/1

The handle and sequence number are still embedded but this identifier is not tied to a particular server or protocol.

This can be used e.g. in a METS packaging/manifest (whether SIP, AIP or DIP) to refer to bitstreams in a way completely decoupled from the home repository. The existing sequence number can be used. Nothing in the DSpace code really needs to change except for perhaps some simple resolution mechanism.

Note that "changing" a bitstream actually results in a new bitstream with a new identifier (even in DSpace today). Actually this is largely a side-effect of how the APIs work but (debatably) it's a desirable property.

(Of course, identifiers may themselves become a more 'pluggable' aspect themselves, but the above relates to currently established best practice and thus desirable "out of the box" functionality. To that end, you might want the bitstream and all other identifiers to contain 'hdl' to indicate the use of handles, e.g.:)

info:dspace/hdl/1721.1/564/bitstream/1

5.2 Bundles

Initially Bundles were meant to be different representations of the same abstract "work" (or maybe FRBR "expression") – e.g. bundle A contains a PDF, bundle B contains HTML + GIF images. It soon emerged that all representations are not equal (you can't assume contents of bundles are "equivalent"), and that other kinds of data need to go in them (extracted full text, thumbnails, metadata). To help distinguish them, bundles were given names, but their use has gradually emerged and changed in an ad hoc way, and is the subject of confusion.

Right now (at least, the last time I looked, things may have changed) we have:

- ORIGINAL, which actually means all “content bitstreams”. In general, few people store multiple (“equivalent”) representations of a single object in DSpace, and when they do, they all go in ORIGINAL. This is more because the UI and batch tools don’t comfortably allow the creation of a richer bundle structure.
- TEXT, which is actually extracted full text. From which bitstream the text came can be inferred by the filename (foo.doc -> foo.doc.txt), which of course breaks if there are >1 bitstreams with the same filename
- THUMBNAIL – auto-generated thumbnails – correspondence with originals as for TEXT
- LICENSE – deposit licence
- CC_LICENSE – Creative Commons licence (in no less than three separate forms)
- METADATA – collection of metadata bitstreams. You’d better hope you’ve some way of working out what metadata is related to which bitstream from the metadata itself.

Clearly better and more uniform practices are needed, and potentially more structure. Also better UI and batch tools – however in this case, the complexity must be very carefully managed.

My suggestion for moving forward is to also give Bundles identifiers using the info: URI scheme. For certain types of bundle with particular content, the identifier can contain the (controlled vocabulary) name – this vocabulary should be carefully controlled. However, since the semantics of ORIGINAL are unclear, I suggest dropping that and using sequence numbers for all bundles that don’t fit into one of the other categories. This means “named” bundles essentially contain metadata, and the numbered bundles contain content (though the line does get blurry at times).

So bundle identifiers in a typical item would look like this:

`info:dspace/1721.1/345/bundle/1`
`info:dspace/1721.1/345/bundle/2`
`info:dspace/1721.1/345/bundle/3`
	Content bitstreams (was ORIGINAL)

`info:dspace/1721.1/345/bundle/current_packaging`	Contains ONLY the latest METS packaging (manifest) bitstream
`info:dspace/1721.1/345/bundle/old_packaging`	Any previous METS packaging bitstreams being kept
`info:dspace/1721.1/345/bundle/thumbnails`	As THUMBNAILS now
`info:dspace/1721.1/345/bundle/extracted_text`	As TEXT now
`info:dspace/1721.1/345/bundle/deposit_license`	As LICENSE now
`info:dspace/1721.1/345/bundle/distribution_license`	As CC_LICENSE now, except single form of the licence, and given an

	appropriate bitstream format that is different from the deposit licence
info:dspace/1721.1/345/bundle/metadata	Other metadata bitstreams

Note "packaging" is used in the bundle names instead of "manifest" to be more consistent with the OAIS model.

Also note that we could decide to put the licenses straight in the METS packaging bitstreams instead of having them as separate bitstreams.

Now that bundles have identifiers, they can be referred to elsewhere in the system and outside. In particular, in the METS packaging and elsewhere it is possible to attach metadata to them, for example, representation information.

This of course does not answer the question of how to enable a rich bundle structure to be exploited. This is mainly a UI issue. The above changes require little or no changes to the existing data model and UI (though there would be a minor migration task to rename bundles and assign identifiers), but the changes open up a variety of possibilities.

5.3 (OAIS) Representation Information

Right now DSpace does have a Bitstream Format registry. The original intention was that this would be a very fine-grained registry – e.g. separate entries for different versions of MS Word and so forth. However, as an initial hack to get the thing out of the door, file extensions were originally used as the means to identify formats. This was supposed to be very temporary but some things turn out to be rather less temporary than others. Still, work is under way to address this using JHOVE etc.

The complexity issue must be managed here again – not everyone will be able to maintain a fine-grained registry. There are global registries emerging and some sort of synchronisation between the local DSpace and global registries will be needed (perhaps via OAI-PMH?)

In the short term, it should be possible to assign identifiers to formats – at present they don't have any sort of identifier and they absolutely need one, possibly more.

A bitstream format might have a PRONOM identifier, a GDFR identifier, and a local identifier. Easy to code, harder to manage, especially when the granularity varies – one registry might have one entry for PDF, another 12 for different versions. If a format entry in my DSpace is for all versions of PDF, that should not share the same identifier as one or all of the finer-grained formats, because they aren't the same thing.

My recommendation would be to give Bitstream formats in DSpace an info: URI, e.g.

info:dspace/(site id)/format/12

The relationship between this and the global registry formats can be then managed later on. (Note "site id" should uniquely identify the DSpace instance. Handle prefix may be used, or perhaps assigning a DSpace repository a Handle itself is a good idea, as a "root" object. Server DNS name is probably not ideal and too fragile.) In any case, the DSpace data model/API can easily be extended to allow extra identifiers to

be assigned to bitstream formats (or rather, a map of global -> local registry format identifiers), which should be used when importing/exporting.

(One could use *only* the identifiers from global registries; however local DSpace might want to have extra formats, or hang extra information off the entries, or have different granularity, hence this local ID + mapping approach).

However, simple bitstream formats aren't enough. Datasets need data dictionaries; .zip files and .avi files have constituent components that have their own format (the compressed files or the audio and video streams).

It's not a given that knowing the precise format of every file and constituent thereof in the system is going to be crucial to preservation – one could instead focus on maintaining tools and data that are good at identifying formats. However, being able to store as rich representation information as possible feels like a direction that we would be remiss to avoid, with the realisation that it will not always be possible or even desirable to expend the effort required to have fine-grained knowledge of the formats.

There are other kinds of representation information as well. The hardware and software environment some interactive object needs to run in; descriptions of the fields in an XML dataset; "attributes" of a format (e.g. character encoding) and so on. It's pretty clear that in many cases, a single bitstream format is just not enough.

Representation information itself is another set of data that need to be managed and are likely to change in format, standard etc. over time. A further complication is that some representation information relates to a set of bitstreams as opposed to just one. However, the new bundle identifiers give us a hook there.

The PREMIS work offers an existing standard to keep this representation information in. I propose taking advantage of that work. In the METS packaging, each bitstream (and, potentially, bundle) can have some attached PREMIS metadata. (Alternatively, this PREMIS metadata could be stored in a separate bitstream and the "format" of the bitstream could be the identifier of that PREMIS bitstream; however this feels like it would lead to an unnecessarily large number of bitstreams over time). The real question when using the PREMIS standard is how much of it is specified in the relational database and how much is only present in the METS packaging.

"Higher level" kinds of representation information (essentially the non-technical sort) is treated as descriptive metadata in DSpace (or perhaps another item, with the relationship specified in the metadata).

(Note: From a "clean modelling" perspective, it might seem nice that each bundle and/or bitstream has a single "representation information" identifier. This could be either a bitstream format or the Handle of the richer Rep. Info. object. However, moving to a more pure Kahn/Wilensky model does introduce undesirable complexity – it could be done at a later date, but right now, I think the ability to have the representation information part of the same object with the same Handle is more desirable.)

5.4 Communities and Collections

The community and collection structure has proved flexible enough to meet most people's needs without being at all confusing. The first thing I'd do with communities and collections is standardise their metadata on DC or some other

standard, rather than having the bespoke, ad-hoc database columns they currently have. A couple of things might be tricky to get in there, e.g. the item template, but a standard should be used where possible (e.g. dc.title, dc.description).

Another easy win would be to enable Communities and Collections to be renamed using the Messages.properties mechanism (have all mentions of "Community" or "Communities" refer to one or two message keys.)

Communities and collections will definitely need METS packaging. This means they could be versioned, and their metadata managed by the DSpace storage system (including checksum checking etc.) (Whether they should also have Bundles is a different matter – probably best to avoid this complexity in the first instance, though later down the line, making DSpace objects internally look as similar as possible may have benefits.)

I can also envisage moving to a more generic "container" model, where DSpace under the hood is more raw Kahn/Wilensky architecture – digital objects contain other digital objects, with some "types". That should probably be done later, and with care to avoid adding complexity.

5.5 Packaging

A short note about terminology: now I refer to the METS "package" or "manifest" as the "METS packaging" because that seems to most closely and unambiguously correspond with the OAIS model.

Back in March 2004 I first introduced the idea of using METS (or some other standard) as the 'native' format in DSpace. This was met with a chorus of "what's wrong with the database" or "we'll lose the transaction-safety of the database". The idea was never to *get rid of* the database, the idea was twofold:

- Get a canonical serialisation of the metadata, including bitstream references and checksums, so that the metadata itself can be easily backed up, replicated, checksummed, etc. etc.
- Provide a place to store more complex kinds of metadata that don't easily fit into a relational database.

I also had a hard time convincing people that treating these METS-based AIPs as the authoritative version of content, and the database as a cache, was the right path forward. The main (and valid) concern was synchronisation.

Maybe the original proposal seemed too radical, or the explanation was too focussed on how it might be reflected in a simple file system implementation or the storage API and implementation in general. I am still convinced (and most of the community now seem convinced) of the validity of the approach; however instead of thinking at all about a storage API, I propose staying with the one we have for now, because it gives us that all-important transaction-safety.

While there are many possible packaging formats (METS, MPEG-21 DIDL, FOXML, XFDU etc) I'll assume METS is chosen to make the description easier below.

I suggest storing the METS packaging as a peer bitstream inside an item. This instantly means it becomes part of the item, is backed up, its checksum is verified, its format (which will likely change over time, just as other bitstreams) is tracked.

Naturally, other bitstreams in the item are included by reference, using their info: identifier.

In terms of synchronisation, I propose that we try generating and storing the METS packaging as part of the same transaction as any update to the database table. This means all updates to both database and METS can be mediated by a single Java object (org.dspace.content.Item), greatly reducing any chances of conflict. This should probably be implemented as a prototype to uncover any performance or other issues. I expect that it should probably be 'illegal' for anything to update the METS packaging directly. In terms of performance, I'm reasonably confident that this won't prove a major bottleneck, particularly if indexing tasks are decoupled from the item.update() call.

The major remaining issues, then, become:

- How to create an API that allows the increased flexibility of the METS packaging to be exploited
- How to deal with the fact that not everything in the METS packaging will also be in the database

There are numerous options here. Org.dspace.content.Item could *only* every update (its internal cache of) the METS document, and the DB tables (Dublin Core etc) could always be derived from that.

One more issue to think about is whether the METS packaging bitstream should contain all of the metadata in-line, or whether it might include some by reference (using the info: URI of the relevant metadata bitstream to do so). Intuitively, it feels like having everything as part of the METS bitstream is simplest; however in the case of non-XML Schema validating metadata, for example RDF, it makes sense to store that in a separate bitstream. Again, managing complexity will be the key.

5.6 Versioning

The METS packaging approach provides a compelling answer to the ever-present versioning issue. Note that by versioning, I mean revisions of a single item, for example when the metadata has been changed, a format migrated, a new bitstream added etc.

If two versions essentially constitute different works (e.g. the descriptive metadata is going to be different because the two versions were initially published by different means), my recommendation remains that each version be a separate item, with a separate Handle and metadata, and that the "later version of" is expressed in the descriptive metadata.

For different versions of a single item, I suggest introducing a new identifier, a version identifier, again using the info URI scheme, for example:

```
info:dspace/1721.1/465/version/1
```

Note the embedded Handle. It doesn't really matter what's at the end; sequential numbers is fine.

Each version is tied to a particular instance of the METS packaging. When an update occurs, a new METS packaging bitstream is produced. Because new bitstreams always get new identifiers, this new packaging will have a new identifier. The METS packaging bitstream should contain the correspondence between previous version

identifiers and METS packaging bitstreams. The most current METS packaging bitstream is always in the "current_packaging" bundle (see above) and older versions (if retained) are stored in the "old_packaging" bundle.

Note that not all versions may be retrievable, depending on local policy. (This could be implemented as simply as a config property or a checkbox on the 'edit item' page, which says "retain old version", with a default from dspace.cfg.)

(Note also that the bitstream identifiers of the METS packaging bitstreams themselves could be used as version identifiers; however I think this would lead to confusion e.g. when 'retrieving' or resolving the identifier – are you referring to the version or the set of bits?)

To make this a bit clearer, here's an example. This gets pretty nitty-gritty, but this complexity should be hidden from the vast majority of users (including admins/curators).

I create an item with a TIFF satellite image and some GIS metadata in XML. Of course there is METS packaging, which refers to both of these bitstreams by their info: identifier.

ID: `hdl:123.456/789`
TIFF image bitstream: `info:dspace/123.456/789/bitstream/1` GIS metadata bitstream: `info:dspace/123.456/789/bitstream/2`
(Latest) Version ID: `info:dspace/123.456/789/version/1` METS packaging: `info:dspace/123.456/789/bitstream/3`

Later, I update the GIS metadata to fix some mistakes and update its format. Now there will be the following identifiers:

I create an item with a TIFF satellite image and some GIS metadata in XML.

ID: hdl:123.456/789
TIFF image bitstream: `info:dspace/123.456/789/bitstream/1` Old GIS metadata bitstream: `info:dspace/123.456/789/bitstream/2` Updated GIS metadata bitstream: `info:dspace/123.456/789/bitstream/4`
Version ID: `info:dspace/123.456/789/version/1` METS packaging: `info:dspace/123.456/789/bitstream/3`
(Latest) Version ID: `info:dspace/123.456/789/version/2` METS packaging: `info:dspace/123.456/789/bitstream/5`

Now we have all of the information about both versions. If we choose not to throw any bitstreams away, we can reconstruct any version given the version identifier. Local policy (which could be configured very simply) can decide whether to keep all of the bitstreams (e.g. the old GIS metadata bitstream) or discard them; and whether to keep the METS packaging for older versions, which would allow you to see how an object has changed, even if you can't retrieve every bitstream.

The main issues and challenges are:

- Maintaining correspondence between version IDs and METS packaging bitstream IDs over time. This could be managed in the database, or the latest METS packaging info:dspace/123.456/789/bitstream/5 contains a reference to the old METS packaging (info:dspace/123.456/789/bitstream/3) and version IDs, and each METS packaging bitstream could contain its own version ID.
- If information about older versions is held in more recent METS packaging bitstreams, and we don't retain every bitstream of every version, there will be some references to bitstreams in the METS that don't exist. This may confuse automated integrity checking tools or any process of ingesting the DSpace AIPs directly into another system. This situation would need to be clearly documented.
- Creating a UI that makes accessing/managing different versions (and retention policies) easy.
- Granularity of versions. Right now, interactions with DSpace usually take the form of HTTP requests and responses in the Web UI. From a 'logical' point of view, a set of those would constitute a single admin action (e.g. add this bitstream, remove that, fix this metadata field, fix that) but it's pretty hard to work out which of the states the object is in during such an operation represents a 'version'. Some session logic may need to be introduced, or some other means of working this out. An alternative overall approach would be to have the versions (and attendant METS packaging generation) happen on a periodic, polled basis, or only get tagged as versions after some period of inactivity, so any if a rapid set of changes happens in the course of a single hour, only one new version is created.

(Note that with the exception of the version identifiers, this is pretty much exactly the approach I suggested in March 2004, but hopefully this explanation is much clearer).

Note that once Communities and Collections have METS packaging, this versioning strategy can apply to them too.

5.7 Provenance

For a preservation-related system, DSpace probably doesn't pay provenance due diligence.

Right now in DSpace there are essentially two sets of data that could be described as "provenance". One is the text in the Dublin Core metadata record. Although this can technically be entered and amended by a user, it's generally mechanically generated, including the checksums of bitstreams, the user who was currently authenticated when the change was made, and the date. The second is the detailed logging information stored in the History System.

The "History System" was named thus because there was some disagreement over whether what is essentially detailed, structured log data was the same as information about the chain of custody of an item.

There are a number of questions to ask here:

- Why the similar data in these two places?

- Should "history" (or better, audit trail) information be considered part of an AIP/item, or should it be captured and managed elsewhere? Which is most appropriate/secure?
- How best to capture the non-mechanical aspects? I.e. what was the user's intent? Where did the object come from prior to DSpace?
- What granularity of event should be captured? This is a similar (or maybe even the same) issue to the granularity issues discussed in "versioning" above. The current History system captures way too fine-grained information.
- When should we start capturing this information? On submit, or commit to archive (full ingest?) The vast majority of the current History System data was captured during the user submission process which is probably useless data.

The METS packaging and versioning approach offers a good way to simplify some aspects. Since a METS packaging bitstream includes the other bitstreams and checksums by reference, it accurately represents the state of the whole item at a moment in time. Thus, these packaging bitstreams (or identifiers + digests/checksums) can be recorded in audit trails/the history system, without the need to serialise a chunk of metadata.

My suggestion would be to radically simplify the History system to use the PREMIS data model, and to store the checksums and identifiers of the METS packaging bitstreams only. If being able to backtrack to previous versions for any purpose is required, the retention policy of the system can be that the METS packaging bitstreams for all (or key) versions of an item are kept.

An alternative is that the history system actually stores a copy of the METS packaging bitstreams. This obviously increases the storage management issues but has compelling advantages.

Exploration of these options and experimentation is probably required.

5.8 An Improved Data Model

Taking into account the above discussion and features, and our data model at the item level looks something like this:

The Handle is still the primary identifier for the item. It identifies the *latest version* of the item; it's the OAIS Content Information identifier.

A *version identifier* uniquely identified separate versions of an item.

Associated with each version identifier is a METS packaging bitstream. The latest is in the "current_packaging" bundle. Older versions (if retained) are in "previous_packaging". These packaging bitstreams can be used as the basis for simplifying provenance-related tools and data models.

Bundles and bitstreams are given identifiers. Note that these identifiers do not necessarily need to be explicitly stored anywhere in the database; most can be examined to find the appropriate object. (e.g. info:dspace/1721.1/234/bitstream/1 can be found using the Handle and sequence number.)

Communities and Collections are also given METS packaging, and their metadata ported (where possible) to proper standards (e.g. DC).

All this is done with the existing, transactional storage system.

The biggest challenge then will be how to take advantage of the improved metadata capabilities with a well-designed API that can keep the relational DB and METS packaging in sync.

6 Modularity

Everyone has always wanted this, which of course is why it featured heavily in my original March 2004 proposals. With good reason: many are struggling to keep their customisations up to date with new DSpace versions, draining valuable resources from the community.

Before blaming the architecture, it's worth pointing out that following good practices can make this *much* easier: e.g. don't mess with the core classes in org.dspace, if you need to alter the database only add new tables, never alter existing ones, and so forth, avoid the temptation to re-invent wheels like database access (even if the existing APIs aren't ideal.) I mentioned much of this on the Wiki page http://wiki.dspace.org/AddOnMechanism. In fact, many "plug in" frameworks are as much about enforcing a set of practices as actual features.

There are numerous existing mechanisms, frameworks and so forth out there for achieving all this – Spring, Excalibur, Java Plug-in Framework, etc. etc.

I think discussing these frameworks is putting the cart before the horse. There are various decisions that need to be made first, and *then* the various frameworks can be examined to find the one that fits best.

The primary needs are:

- Enable adopters and researchers to develop, maintain and distribute their customisations independently of the core, not needing numerous complex "merge" operations
- Enable the functionality of DSpace to vary from deployment to deployment to suit local needs
- Better support a "divide and conquer" approach to maintenance and development so that different teams and individuals can take responsibility for clearly delineated areas of functionality

The tricky parts of realising this have always been:

- The relational database. These are optimised to be rather static structures. Updating, independently-managed "sub-schemas" is difficult (and if modules actually try to insert new columns in existing "core" tables, the situation becomes untenable). This is a "best practice" issue, though it could be supported/enforced by various mechanisms.
- People keep editing core classes, adding new methods or opening up private/protected methods. This may or may not be necessary to make a particular piece of functionality work, but I get the feeling people haven't done due diligence to isolating their changes to e.g. the Web UI. Again it's a best practice issue that can be supported or enforced, for example by actually

separating the core code from the application/interface code in CVS so that people have to submit patches to each separately.

- The Web UI JSPs. Adding some new UI functionality generally involved editing a few bits and pieces of JSPs, to add links here, buttons there and so forth. Manakin should help address this. It feels like what people are really after is a kind of portal-style UI approach to DSpace.
- Working out what the appropriate APIs and interactions are, especially what's "core". I made a vague stab at this back in March 2004, but people got caught up in discussions about the means rather than trying to work out what the modules and interfaces should be. But this will be much, much trickier than selecting which uber-framework to use.

The plug-in manager and add-on mechanism are steps in the right direction. For the add-on mechanism, my strong recommendation is to get something simple that improves the current situation out as soon as possible, instead of trying to boil the ocean and try and make it "the" modularity answer. There will always be limitations to what any modularisation mechanism will let you achieve: live with it.

There are further aspects of modularity that should be examined before a framework/approach is decided:

- Network vs in-process APIs. DSpace interacts with other systems via network protocols. Potentially, network APIs could also be used by DSpace internally, so that different modules could run on different machines. (Usually referred to as Service Oriented Architecture, specifically meaning Web Services.) This approach has its own issues, particularly in terms of complexity of deployment, and performance under heavy load as each component requires network resources to access others, and many network interactions may be needed to meet a single user request. It could well be that a cluster of "parallel" servers works better. A further issue with network APIs is that right now, there really aren't any suitable standards for many of the kinds of interface and data interchange we need. We wouldn't necessarily be opening DSpace up to external systems in the way we'd like; we'd just have a set of proprietary DSpace APIs that happen to be network-based rather than in-process Java API based. So, although I'm not saying it should never happen, using network APIs *inside* DSpace is a step that should be thought through *very* carefully.
- "Supported" versus non-supported (user-contributed) modules, as discussed in section 2.3.
- Trade-offs between clean separation of modules, complexity, and performance. This will be a tricky thing to balance. Total separation of modules (e.g. no shared database access at all) is likely to have an impact on performance under load by increasing the amount of data being moved around. Less than clean separation probably increases complexity of management and so forth.
- Versions of each. Certainly for the core APIs, and probably for others, it will be necessary to strictly version them, and maintain compatibility matrices. Upgrades will also be tricky. What happens if you upgrade the core but an application module stops working? What happens if a core upgrade causes

some database schema migration that triggers constraint violations in a module's (separate) database tables? What order should upgrades be done in? Again, frameworks like Spring etc aren't likely to fix this – it's down to establishing and following a set of practices and that will take management.

6.1 Publish/subscribe

I'd hate to talk about modularity without making at least one concrete suggestion. One of the first ideas in the "DSpace 2.0" proposal I made in March 2004 to attract animated criticism was that instead of managing the complexity of publish/subscribe messaging (with attendant issues around robust message delivery etc) we take a polling approach to allow decoupled modules to keep in sync with the asset store. I still think there's merit in that approach, and certainly it's the only approach I'd consider for any widely distributed system (like the China Digital Museum project).

However, the other approach, despite its complexities is still worth pursuing. (And as I point out in http://wiki.dspace.org/RobsAssetStorePrototype each can be built on the other).

To that end, I'd suggest a publish/subscribe model be introduced, initially just for item (or maybe 'object') creation, updates and deletion. Other event types could be added later.

This achieves two things:

- Modules interested in getting updates can be decoupled from the core code, without for example having to insert code into Item.update().
- Processes currently that happen in serial with item updates (including batch ingests) can be run in parallel on another CPU/server, or at a lower priority, or otherwise scheduled. People have run into problems with batch ingests taking a long time. This is because DSpace is going ingest, index, ingest, index, … and if the index doesn't finish quickly the whole process gets bogged down.

Of course, the tricky part is that this introduces significant complexity. The system suddenly has internal state – messages need to be reliably delivered, a broker will probably will need to be running, and components will need a initialise/terminate lifecycle, making things like crash recovery more difficult; clustering servers may become *much* more difficult.

That said, all this stuff is essentially a solved problem, with existing best practices and technologies (including open source implementations) for dealing with this. If we can build the system in such a way as to hide these complexities from most "lightweight" users, this could be a good route forward. One could envisage other areas of the system benefiting from a publish/subscribe mechanism, for example it could form an element of the ingest workflow system.

When choosing a technology for this (I'd suggest "rolling our own" *definitely* is not the way to go), we'll need to decide whether to go for a Java-based approach such as JMS/ActiveMQ or a more platform-neutral approach based on WS-Notify etc.

When it comes to addressing that transaction-safety, the Context object will continue to be crucial. Instead of instantly despatching messages, the messages can be lined up in the Context object. The messages can be fired off only when the Context object is successfully committed.

One more tricky issue is to do with a potential "infinite loop". If a subscriber changes an item when they hear it's been changed, that will trigger a further "item changed" message. E.g. if I have a subscriber that adds a provenance metadata field; someone changes the item, meaning an "item changed" message is issued, which means the provenance is added, and another "item changed" message is issued. Will need to watch out for this one.

7 Authorisation

Clearly many issues here, in various dimensions:

- "ResourcePolicy" table needs renaming – they're permissions, not policies
- Granularity of permissions. READ, WRITE, ADD, REMOVE don't seem sufficient. E.g. does not having READ permission mean I'm not allowed to know of an object's existence?
- Conflation of roles and permissions. Some of the "permissions" or "policies" in that ResourcePolicy table are really roles, e.g. workflow step 1, admin etc. This makes management very difficult. Separating roles from permissions (and maybe having policies as a further separate entity) may greatly simplify things.
- Management tools. We really need good interfaces and other tools (e.g. to integrate with personnel/roles DBs) to make managing permissions easy, regardless of the underlying implementation. This will be difficult, as there are tricky issues like inheritance of permissions and default permissions for new objects to take into account.

There are options for taking "off-the-shelf" technologies like Shibboleth, but these might not be appropriate for all. What is probably needed is a good conceptual model (role-based) and an API or set of APIs that enable different implementations to be used for different organisations.

8 Collection Management, Curation, Administration

Digital preservation is a process, not a technology. I'm not quite sure where claims that DSpace is "OAIS compliant" came from, but since OAIS talks about processes, communities and responsibilities, DSpace itself can no more be "OAIS compliant" than a set of pliers can be a certified electrician.

In addition to the technological means needed to assist the process (e.g. checksum checking, provenance), collection managers and curators (who will probably not be system administrators) need tools and interfaces to carry out the preservation processes, as well as perform other administrative functions like manage authorisations and users, orchestrate federation/replication agreements and so forth.

DSpace does have some collection management/curation functionality and it is a key differentiator for the DSpace system, as well as being a critical area for research. However, as of yet, it hasn't received the attention it needs.

The critical areas are:

- The edit item form. It's pretty hard to use, and doesn't capture any provenance information. It's only one step away from hacking the database directly with SQL. It needs a lot of work.
- The authorisation administration UI – it's pretty awkward to see what's going on, and who has what permissions, as described above. I don't know how you'd tell if there was a glaring security hole.
- Strongly related to the above, epeople and group membership
- Managing archive consistency, deletions and so forth
- Capturing provenance for admin operations

There's enough room for a slew of research projects here.

9 Internationalisation

It's come a long way, but issues still remain in various areas. Some contributions may address some of these:

- Emails
- Users being able to select interactively, and their preference stored for emails etc
- Descriptive metadata – the database + APIs can capture multi-lingual metadata, but the UI doesn't comfortably allow this. Perhaps an extra "initial question" in the submit UI ("the metadata is in multiple languages" or some such)
- Content – can only have one item-level dc.language field. What if there's a French PDF and a German PDF in one item?
- Browse: the browse controls ("A-Z" doesn't work for everyone) and ordering
- Search: multilingual search. I asked on the Lucene list a while ago what would be the best approach, see:

http://mail-archives.apache.org/mod_mbox/lucene-java-user/200505.mbox/browser
http://mail-archives.apache.org/mod_mbox/lucene-java-user/200506.mbox/browser

Approaches include:

- A single index
- Different analyzers/indexes for each language
- Language being a field in a (Lucene) document, and a separate document for each language
- Each field name in the Lucene document being of the form fieldname_language

No clear best practice emerged, but people mentioned success using a single index with an extra field for language.

- On-line help

- System documentation etc
- Installation and sysadmin script output
- How to manage all this when there are many different modules
- Managing "localisations" during updates, perhaps having a local 'override' file that overrides message keys but doesn't specify those keys where the default is OK

10 Scalability

I've mentioned this several times in the preceding sections. Any specific questions about DSpace scalability (how many objects can it store, how many users can it serve) don't make much sense, it's kind of like asking "how many files can a hard disk store". It of course all depends on the hardware and storage. It's not really reasonable to expect to run an enterprise-scale system with millions of objects and terabytes of storage on a single server. DSpace can be run on a cluster and can be made to scale to manage numbers of users as required.

There are various dimensions of scale to consider:

- Number of concurrent users
- Amount of storage
- Performance with large numbers of objects (indexing, ingest, search/browse response times)
- Size of individual objects (e.g. 100Gb data files)
- Administrative and curatorial workload (number of FTE/number of objects)
- Barrier to deposit vs quality of deposits (easy to get lots of bad content)

There are clearly bottlenecks in the system that should be addressed, however.

- The browse system indexing performance in particular degrades badly as the number of objects increases. As described above, decoupling the indexing process from the batch ingest process would also help.
- There has been talk of optimising by looking at the database schema: Some profiling work by John Hopkins University amongst others uncovered the worst issues. The focus should now be on ensuring that the code itself does not issue more SQL queries than necessary: I suspect that far more queries are being issued than necessary.
- The database access API/mechanism could be refactored to make use of PreparedStatements.
- The in-memory object cache of the Context object can get rather large, hogging memory. Weak reference hashmaps and "lazy initialisation" help but more may need to be done.
- More caching in general could be done at the UI layer. This is pretty tricky in the pseudo-MVC Servlet/JSP environment – you can cache JSPs but since most of the work is done in Servlets, you don't gain that much. Hopefull

Manakin can help. Also other things can help like sitemaps or static pages optimised for Web crawlers to minimise their impact.

- The OAI-PMH harvesting could do with optimisation, perhaps by caching results.
- More in-memory caching of objects in general may help. However this might be tricky to achieve with the Context object mechanism (which has proved immensely useful in other aspects).

11 Storage

In this document I've expressly separated issues of data model, AIPs, METS packaging etc. from storage – I think their conflation in my original DSpace 2.0 proposal and the ensuing prototypes was a big problem.

In general, the current DSpace storage is fine in that it's transactional. However, the initial, minimal hack approach to having several asset stores has remained and not been fixed. Java's current lack of a way to find out free disk space is an issue there, but I believe that is being addressed in a future Java version.

Grid-style storage offers a great potential to scale up DSpace storage. The existing SRB storage mechanism is OK as a proof-of-concept, though ideally should be separated out from the file system-based storage code as a separate plug-in/module.

It's another balancing act as to how much DSpace is responsible for managing storage and policies versus the underlying technology (which is likely to vary). E.g. SRB will be managed differently from LOCKSS, ChinaGrid, or some off the shelf HSM system (online, near-line, off-line). DSpace could give storage "hints" or policies through some neutral, standard language which could be acted upon by the underlying storage technology (or management tools thereof). The key is to keep DSpace simple so that it can work with a simple file system, but to enable it to take advantage of the various storage technologies available. SRB is just one answer, and is not fully open source so alternatives should be available.

11.1 The Relational Database

I've grown a little concerned at the over-use of the relational database, even for things like constants that could happily live in Java class files. New relations and columns should be added to the database only when necessary – managing schema upgrades and so forth is a big challenge. This is particularly true where the volume of data will grow quickly, for example usage logs.

12 "Standard" metadata

The Dublin Core profile that DSpace uses is based on a very early draft of the Library Application Profile. This contains several non-standard qualifiers, including of course the choice of "contributor.author" which has ruffled so many feathers. It's a fair point – DSpace, at least in the "out of the box" configuration should probably accurately follow the latest standard. For existing users there's potentially an "upgrade

crosswalk" that might be required. Really the metadata, just as the content bitstreams, might be subject to preservation migrations as standards evolve; that's one reason why having the METS packaging as a bitstream in the item is so appealing.

Anyway, the specific problems in the use of DC in DSpace are:

- Use of non-standard qualifiers, which make expression in XML difficult
- Some fields relate to bitstreams as opposed to the item level, and in multi-bitstream items this means they are meaningless. For example, items may have several format.extent and format.mimetype elements, with no way of corresponding values to individual bitstreams.

Of course, this is compounded by people inventing their own qualifiers etc. That's what the "multiple metadata schema" mechanism was intended to allow – use DC or other standard namespace metadata values where you can, and if you have to invent something, at least do it in your own namespace so you can tell it's not really DC. However, this needs people to have the expertise and will to enforce this, and tools to make changes in this area don't yet exist.

13 Dissemination Architecture

There has long been talk of "dissemination architecture" for complex objects, on-the-fly migration/conversions, integrated streaming servers and so forth. This is another balancing act – how much into "publishing" should the DSpace system itself get? I don't think decoupling access from storage/archiving and collection management is necessarily a good idea from a digital preservation perspective. Unlike in the physical preservation world, constant human access of the digital material is a good thing because people will uncover problems that automated processes might not.

14 Automated Testing

It's often been pointed out that automated testing (or unit or regression testing) would greatly ease the contribution inclusion process by automating part of the evaluation process. This is definitely true. It's worth pointing out that automated testing can be achieved without necessarily refactoring the code or architecture (and let's get away from this "we'll fix it in version 2" syndrome which has meant various hard problems haven't been looked at.) There's no reason we can't do this with the current architecture. I think a 'test' corpus, with diverse enough content and metadata and unencumbered by restrictive usage rights is the first requirement; once this is in place, it should be possible to write lots of tests to give the org.dspace.content and other APIs a workout given that you know the expected behaviour. It might not be fast or pretty as the corpus would need to be 'reset' each time, but would help with our QA and patch testing processes immeasurably.

15 Code Management

There are various issues or areas of concern regarding management of the code itself.

- SourceForge has served us well thus far, however as we search for ways to better support the distributed developer community we might want to examine other options. What SourceForge doesn't comfortably let us do is give different groups different permissions on parts of the source tree. Such a capability would let us set up focussed committer groups with responsibilities for particular aspects of the system. Also it would let us set up areas for members of the community to use for their own projects, making them available to all and freeing them of the burden of running their own local development infrastructure.
- The code really needs regular "housekeeping" or "vacuuming" to look for parts which need cleaning up, documenting, fixing up. The biggest problem here is motivating people to do it – people like to build new stuff, not fix others'. http://wiki.dspace.org/CleanupTasks lists some areas.
- Maintaining consistency. Very tricky in a big, distributed open source dev community. However the source of the problem is also the reason why consistency is so critical – we need there to be one clear way to access the database, do authorisation/transactions, do logging etc. In some areas this has started to 'creep' as multiple ways of doing things emerge. These should be cleaned up over time, as they will lead to confusion and less manageable code over time.
- Platform independence. Really this relies on enough active people using the given platform (be it Windows/MacOS or Oracle) to be able to report and fix problems and keep things in sync. Various steps can be taken to mitigate the difficulties. As far as is possible, vendor-neutral SQL should be used. I also think we should remove as much Perl etc. as possible, not because it's a poor language, but to reduce the number of languages that a DSpace developer needs to know and that a platform has to support.

16 Ingest and workflow

DSpace has a pretty simple ingest system with a very simple submission workflow. A more pipeline-oriented approach would be good, but care should be taken not to make this *too* generic – we did at one point have a very flexible, generic workflow engine but it proved impenetrable and unmanageable by anyone other than the guy who wrote it.

We definitely need better format identification and verification, the ability to automatically extract metadata (both technical and descriptive), do things like create and store digital signatures for provenance reasons and so forth. Ingest processes are likely to vary greatly, so we need flexibility here, but always bearing in mind the "lightweight" users who shouldn't be overwhelmed with complex options.

17 Logging and stats

Way back when, we decided on a particular format for log files for better analysis and statistic collecting. The basic form is:

```
email-address:session_id=XXX:ip_addr=YYY:<action>:<parameters>
```

<action> is intended to be a controlled vocabulary, and parameters a comma-separated list of key/value pairs that correspond to the action. This notion seems to have been lost somewhat, but I think it has value. (Again, the Context object is relied upon). Knowing exactly where to but WARN, INFO and DEBUG-level statements is of course rather difficult, but critical to debugging and statistics.

With statistic-generating, I'm a little concerned about the idea to have loads of statistics stored in the database. As mentioned above, such data will tend to grow and grow, and become a management burden (both for developers and sys admins) over time. My own preference would be for a more regular log analysis approach, where existing log rotation/management tools can be used. Many tools have incremental analysis mechanisms so that it isn't necessary to process all of the logs to update statistics. However, it's not a huge issue either way; there are more pressing issues.

18 Deployment and configuration issues

Deploying DSpace is pretty easy (despite the occasional grumble on the lists). Managing configurations is a little trickier. Once you install one version, you already have to sets of config files – in the source tree and in the installation. When you download another version, you have three, which are probably out of sync.

Maybe this isn't as much of an issue as I think, because there haven't been too many problems reported on the list. However as the system becomes more modular it might become an issue. Do we want a single dspace.cfg file or an archipelago in a directory? Are property files OK or do we want to move to XML? (XML in general seems more suited to computers than people, though friendly tools could be build on top of them.) To what extent should defaults be embedded in the code? Should we always have a fall-back "dspace-default.cfg" that is part of the source code and never changed, with a local dspace.cfg file being essentially a set of diffs? There are numerous options here, and probably examples in other systems to look at.

19 Conclusion

Due to time, this got a bit hurried in the end. But that's probably OK, because I think I focussed on the big problems first, particularly the data model one.

I'll finish with some procedural and structural recommendations about how to move forward.

There are currently two really critical areas where DSpace needs to be taken to the next level: the data model to support more complex metadata and objects and versioning, and modularisation. Following that, the critical issue is improving collection management and administrative UI capabilities.

I think two surveys need to be done, resulting in a summary report:

- A comparative summary of existing data models and packaging formats – MPEG-21 DIDL, XFDU, METS etc.

- A comparative summary of existing modularisation mechanisms, frameworks and so forth.

These could serve as input to two further activities/groups:

- A data model survey group, to look at the requirements for DSpace, and make a recommendation looking something like what I've written above but obviously going further and more concrete
- A modularisation report group, looking at the needs of DSpace, identifying what the key APIs are likely to be and how they should interact, then recommending an approach, possibly employing one of the mechanisms/frameworks

These two activities could then provide valuable input to the main technical working group. A further project looking at the collection mgmt UI would be a great idea.

As I mention in the section on Revolution vs Evolution, I think some focussed, and centralised projects will be needed to do the heavy lifting on these activities. "Volunteer" and contributed developer resources will be extremely helpful in validating ideas, quality assurance and so forth, but to make the progress we need, co-located teams may be essential. There is funding out there for this – JISC, NSF, Mellon, the EU, DLF, financial contributions from adopter organisations.

I remember a long time ago saying that if DSpace is successful, the initial 2 year development effort will be a very small part of DSpace's life. I hope that will be true, and I hope I've done my part to make it happen.