ELSEVIER

Computer Networks 39 (2002) 577–588

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

The design and implementation of the Redland RDF application framework

David Beckett

Institute for Learning and Research Technology, University of Bristol, 8–10 Berkeley Square, Bristol, UK BS8 1HH

Abstract

Resource description framework (RDF) is a general description technology that can be applied to many application domains. Redland is a flexible and efficient RDF system that complements this power and provides high-level interfaces allowing instances of the model to be stored, queried and manipulated in C, Perl, Python, Tcl, Java and other languages. It is implemented using an object-based API, providing several of the implementation classes as modules which can be added, removed or replaced to allow different functionality or application-specific optimisations. The framework provides the core technology for developing new RDF applications, experimenting with implementation techniques, APIs and representations. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: RDF; Metadata; Semantic Web; Application framework

1. Introduction

Resource description framework (RDF) [1] is a general purpose technology that enables the description of resources on the Web using URIs for identifying the resources and URIs for the properties that describe the resources. This design means that there is a large range of applications that can use this generality customised for their target domain. RDF and RDF Schemas [2] are designed to work across domains and provide common facilities for describing resources, collecting them in containers and maintaining type and class relationships. The RDF specifications make very few restrictions on the kind of information that can be recorded in order to provide an open description technology for the Web and thus the software systems that implement them need to be very flexible.

2. Requirements for an RDF application framework

Applications of a general description standard such as RDF have a wide range of needs and ways that they would use the technology although the information is processed, manipulated and stored using the same RDF model. Thus there may be substantial differences between application requirements of an RDF system, and any one implementation of such a system. In order that this interaction be made more efficient, a way was needed of optimising how the RDF system internals worked, depending on the application. This

E-mail address: dave.beckett@bristol.ac.uk (D. Beckett).
URL: http://purl.org/net/dajobe.

led to the need for more than just an implementation of RDF, but a framework around the RDF model that could be flexible enough to optimise for particular applications by providing modules with different implementations.

As of early 2000, the major deployed applications of RDF were mostly either embedded inside products such as the Mozilla [3] Web browser or as separate systems like SiRPAC [4] as used in the GINF project [5] and elsewhere. Mozilla uses RDF extensively for representing the internal platform data sources, as well as for configuring the user interface. This code in C++ is quite integral to Mozilla and difficult to separate out in order to reuse, since it is optimised for Mozilla's object and class systems. SiRPAC is a Java application that grew from a parser into a more general application from the GINF project. SiRPAC is easy to use in Java applications but that is not the language used by most Web applications and hence not suitable for all uses.

The APIs provided by Mozilla, SiRPAC API and other API proposals such as RADIX [6] share a core similarity in the types of concepts that they present, although all of them do so slightly differently. See [7] for a more detailed review made in early April 2000.

There were other well-deployed applications and services that used RDF internally such as rpm2html/rpmfind [8] (and related tools) used extensively for indexing Linux RPMs, and the UK Mirror Service [9] which uses RDF for mirror and content description, however, these mostly use the tree-based XML DOM interface in custom ways for their application rather than present any general RDF interface. This also means that they did not use a full expression of the RDF model or syntax and more specifically did not have full RDF parsers.

RDF applications on the Web or applications that wanted to have RDF support needed more open libraries that were portable, easy to configure, build and integrate into the application. This meant that there was a need for a self-contained, complete and industrial-strength library for RDF that could easily be used with existing applications, and had good integration capacities via APIs in major languages.

RDF requires XML for the syntax and since XML is now a family of technologies that need to be processed, this can be somewhat of a barrier to handle while also dealing with RDF, all in one application. A toolset that presented a higher level interface at the semantic level above XML and RDF syntax would allow applications to work in the concepts of the RDF world rather than get stuck in the detail of XML.

The RDF schema [2] was only recently a candidate recommendation at the time Redland was begun. It was unclear if it required any new special API support or concepts and if these new concepts were more generally useful. It was useful to provide an implementation that could experiment with these APIs and concepts in order to determine such requirements.

These needs mean there was a requirement for a new system implementing a high-level interface for the RDF model that was designed to be portable, integrate with applications written in many languages, be modular so parts could be replaced, provide hooks for research on RDF itself, have sufficient stable interfaces, take on board existing best practice, using standard programming metaphors so that it could be used in different ways and be a solid and industrial-strength implementation.

3. Detailed design

The RDF model is defined in the *RDF model and syntax specification* [1] and unfortunately there is not sufficient space for a full introduction to the model in this paper, but such an introduction can be found in [10]. In the formal description, RDF consists of a collection of statements, which contain three parts (also called a triple or tuple):

(1) Subject—what the statement is about,
(2) Predicate or property,
(3) Object—the value of the statement, which can be a literal string.

Each of the parts of the statement (except for literals) can be identified by an URI allowing statements to be written about any resource with a URI. Predicates are also identified by URIs thus new descriptive properties can be *defined* on the Web, as well as *describing* things on the Web.

Although the statement collection is the formal description, this can also be represented as a graph of nodes (subjects) and arcs (predicates) pointing to other nodes (objects) or literals and this is an easier way to think about RDF—it is a Web of statements.

Redland needed to represent all the concepts in the model and some additional ones including an expression of the collection of statements—called a model in Redland, after SiRPAC. The Mozilla RDF API also includes concepts such as a *DataSource* which are similar to the Redland model and represent a source of statements, and a *composite DataSource* which contains a set of *DataSources* and allows operations over them, as if they were one *DataSource*. Mozilla and SiRPAC both have similar interfaces for de/serialising models to and from a sequence of statements as well as dealing with the RDF XML syntax parsing. SiRPAC has additional facilities providing Java Interfaces and common Java metaphors such as producer/consumer for RDF and Enumeration. Both RDF systems query the model in the same two ways—asking for matching statements, called statement-centric, or dealing with the model in terms of a graph and asking questions relative to a node or arc, called node-centric. There was no other query language syntax defined or consensus how results would be returned from such a syntax.

RDF needed to be stored in a way that accommodated the general case but was efficient for the kinds of queries that particular applications might need. This was an area of research that might require several storage implementations for different purposes. The storage also needed ways to be able to use existing systems such as relational databases. This has been investigated previously in [11] without a single database schema emerging unanimously as the best answer; not that this was unexpected for such a general application.

It was expected that RDF would provide services delivered via the Web. These services may be not be on the same system as the application so support of remote RDF models that were manipulated or queried via what might be called narrow interfaces was required. For example, the only interface might be a request/response query over the remote model and the result would be a sequence of statements that matched, these statements forming a model of their own, or representing a sub-model of the remote one.

Conversely, on systems where it is efficient to represent many models and sub-models in the same storage system, the results of queries might be best represented as models in the same storage. For example, if a relational database is made visible as an RDF model, and a query is performed over it, it is not necessary to create a new stream of statements for the resulting model since the relational database can efficiently express this as a view on the queried model. This gives the requirement for support of model to model operations without the need for serialising them.

Applications of RDF that were being designed and developed at this time required support for provenance tracking. This is expressible in the standard RDF model and it was not clear at that time if it needed special API support, so this became another potential issue for investigation.

In summary, the detailed requirements as derived from the analysis of the model, existing and future application requirements were

- Statement/graph arc;
- Statement parts—subject, predicate, objects/graph nodes;
- Model, aggregate model;
- Storage for models in memory and persistently;
- Parser for the XML syntax;
- Streams of statements for de/serialising models;
- Lists of statement parts for walking RDF graphs;
- Querying with flexibility on query language and how the results are returned;
- Interfaces for models with statement streams and model-to-model;
- Industrial strength and solid system;
- Use existing best practice in APIs, interface and implementation;
- Modules providing different implementations of functionality;
- Any support that may be needed for provenance tracking.

4. Design—patterns and implementation language

This section provides an outline of the design, concepts and programming models that were used to meet the requirements and the detailed design.

4.1. Layering

Redland implements the RDF model, however, this might mean different things to applications, which understand more or less of the detail of the model. Some applications require only certain parts of the functionality and want to do the rest themselves while others want to use Redland as a closed system. These issues are addressed partially by modules—see Section 4.3. In general, though, Redland is part of a layered system with the application at the higher layers, in which Redland might provide multiple lower layers, layers that the application could interface to. It might be useful to provide, for example, a schema-checking model rather than one with no validation. This is performed by layering inside Redland where the schema-checking model uses a lower layer model implementation in order to provide this. Other examples of higher layers that could be provided include models providing associative or bi-directional properties for application specific purposes, checking trust models, digital signatures or with transaction support.

Redland was designed to cover approximately the lower four layers of the building blocks of the diagram shown in Fig. 1, based on one by Berners-Lee [12].

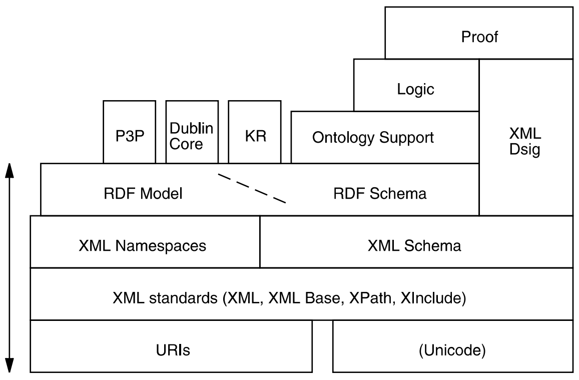

Fig. 1. Redland provides RDF building blocks.

4.2. Objects

The target languages for using Redland were C for use in compiled applications and languages used in many Web applications including at least Perl and Python. This meant that the design had to be appropriate for object-based (Python), non object-based languages (C) and ones that can do both (Perl). The object model is a clean way to specify and implement the design, especially with the requirement for modules (see below) and could be called from non-object-based languages if done carefully. Note that full object-orientation with inheritance is not proposed here (since that is very hard to express in languages like C) but an object-based API using objects, methods and polymorphism.

C++ was an alternative implementation language but it has less support in interfacing to scripting languages, is used in only some applications, and is rather complex. It does provide powerful techniques and support for memory management, OO and libraries that could have been used but some of these would have been difficult to express in non-OO scripting languages. The choice of C and C++ and its consequences on the implementation are discussed further in Section 6.10.

4.3. Modules

The flexible architecture required that there were parts of the system that could have multiple implementations for the same interface. It was also desirable to be able at some point to potentially support dynamic loading of modules at run time on demand or automatically by the system following a specific application request. This suggested the use of the factory pattern where modules can register/de-register with factories at any time. The factory need not be visible at the application level with wrappers around them made via the object constructors.

4.4. Portability

Portability was a major requirement so the system had to be written in C, since virtually all major languages have interfaces to C and indeed they are mostly all written in it. This unfortunately meant a lot of support that is provided by some of the target languages had to be implemented internally. Redland uses C function naming conventions to provide the routines for the constructor, copy constructor, destructor functionality as well as the general methods.

4.5. Interfaces and implementation

In languages like Java, there is a clean interface/implementation separation but using C this had to be emulated by conventions. A Redland class is defined as a public C typedef struct representing the class and its public interface (constructor, etc., methods) defined in a header file along with any public or private types, enumerations or constants. The actual class implementation definition (C struct that the typedef refers to) and internal definitions are not exposed to applications and are only available internally when Redland is compiled. The implementation of the class is defined in a C source file and can include private static functions either for internal implementations or to satisfy part of a factory API.

4.6. Class initialisation and termination

Classes may need to be initialised at load time by what are generally called static or class initialiser code. Redland classes may have a class initialiser/termination pair of functions which must be called before any object in the class is created, and after the last object has been freed. This is needed for many classes but especially those that implement modules which need to be registered at load time, so that they are ready to use when the application code starts and can be de-registered when the application terminates.

5. Redland architecture

This section provides a description of the architecture that was used to implement the detailed design and requirements using the design patterns.

5.1. Redland classes

The requirements, design and language having been chosen, the classes listed in Table 1 were defined for Redland covering the required concepts and the support facilities needed. These were strongly influenced by the SiRPAC Java interface [13], although flattened to reduce the number of classes and extended for the classes that are provided internally by Java such as URI. At the time of writing (November 2001) the *Query*, *WWW* and *Serialiser* classes are not implemented.

The classes are used and associated with each other as shown in Fig. 2. The support classes are used throughout the rest of the classes as needed. The *Stream* classes are used whenever a sequence of statements is accepted or generated by the *Model*, *Storage* or *Parser* classes. The *Model* class uses Stream for performing the serialising/de-serialising the *Model* and returns lists of statements from queries. The *Parser* class only uses it to provide a sequence of statements as the result of a parse.

At the simplest level each *Model* object has a one-to-one mapping to the *Storage* object that represents it. The functionality for aggregate *Models* is present in the *Model* class so that higher level *Models* can have sub-models and in that case there

Table 1
Redland classes

Redland Class	Purpose
Node	RDF graph nodes
Statement	RDF graph arcs (statements, triples) [isa *Resource*]
Model	Set of *statements* held in *Storage*
Storage	Storage for *Models* (modular)
Stream	Sequences of *statements*
Parser	Syntax parsers (modular)
Serialiser*	Serialises a model to a syntax (modular)
Query*	Query syntax adaptor classes (modular)
Iterator	Enumerating lists (of *Node*) from queries
URI	Provides URIs for *Resources*, *Parsers*,...
WWW*	Resolves URIs to get content from the Web
World	Redland environment startup/shutdown
Digest	Content digests (modular)
Hash	Key: value maps with duplicates (modular)
List	Provides support for lists in C

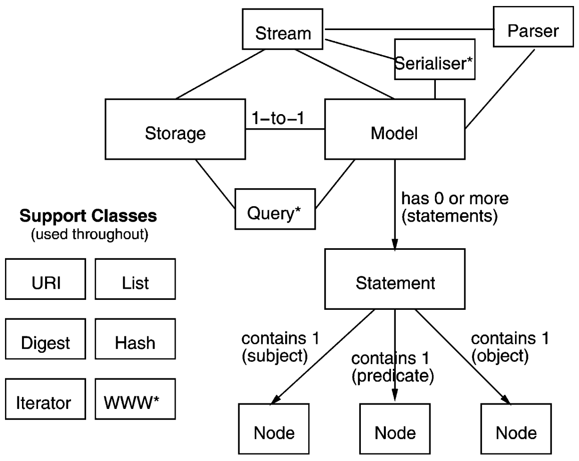

Fig. 2. Redland class diagram.

is no one-to-one mapping to a *Storage* but the higher level *Model* will have a set of sub-*Models* or some other relationship. For example, a higher level *Model* representing a remote information resource may not have a *Storage* but use some other way to present the *Model* interface.

5.2. Modular classes

Each of the modular classes has an internal factory that allows the module implementations of the class to register/deregister themselves. The factory creates these classes for the application via the constructor for the modular class. The modules can have optional implementation of methods for the factory so that the factory can either do the default action itself or implement it another way.

The *Digest* class is provided to allow several message or content digest algorithms to be used with some of the RDF concepts in order to be used for applications like digital signatures or computationally generating identifiers from say, literal strings. The digest classes provided were MD5 (always), RIPME160 and SHA1.

The *Hash* class abstracts key—value mapping (with duplicates) which can be used for many purposes including storage of statements in persistent hashes and other internal uses. The implemented hashes were in-memory (always), GNU DBM hash (optional) and versions of Berkeley DB (BDB)/Sleepycat DB.

The *Storage* class abstracts the storage of models—the *Model* class passes on most of its methods to its associated storage. The implementations of the class currently include an in-memory one (always) and one that uses multiple *Hashes* to store the *statements*, either in-memory or persistently via BDB. This implementation is discussed in more detail in Section 6.2. This class could have been called database or datastore but that would have been confusing since only one of the implementation modules of the class might have been a true relational database.

The *Parser* class provides a common interface to modules that parse various syntaxes to deliver an RDF model. The *Serialiser* class does the reverse and generates syntax from a model. See Section 6.4 for more detailed information on the issues with these classes.

The *Query* class is an adaptor class that provides support for particular query syntaxes for *Storages*. It takes a query as a literal string or as a *Model* along with a *URI* to identity the query language. This *URI* allows the query class to determine if either there is an adaptor class for the language or the storage natively knows it. In the former case, when the storage module does not understand the syntax, the adaptor class rewrites it into a standard query for matching *statements* and submits it to the *Storage*. In the latter case, the query, which need not be written in a statement-centric way, can be directly handled by the *Storage*. This makes the query process a lot more efficient since the query does not need to be rewritten, several layers of system are skipped, and the results do not need to be rewritten as *statements* but can be delivered in a *Model*. This is possbile since the application and the storage module both understand this special optimisation, represented by the *URI*. For example, the query could be in an SQL-like syntax such as in [14] and a relational back-end could handle it very easily without the need to rewrite the syntax into a statement-centric format and thus loose the chance for query optimisation. At the time of writing (November 2001) the *Query* class is not implemented and only the statement and node-centric queries can be performed.

5.3. Data flow and flow of control

Data flows inside Redland mostly from arguments passed via method calls into the implementing classes and possibly onwards to factories and modules. However, when Redland is connecting objects that are more naturally both working in parallel, such as a parser and a consumer of the statements generated, some other abstractions are needed. *Stream* and *Iterator* are used to provide this transfer of data and flow of control for sequences of *statements* and *Nodes* respectively. These classes are reader driven or pulled, since Redland is intended to be used as a library inside an application which will generally be calling Redland to read data rather than pushing data for Redland to process. Redland does not do much processing in one go (apart from parsing) so the pull model is quite natural to use in this way.

When parsing the syntaxes, most of the current parsers need to be active, pulling data from their data sources (files or *URIs*) and so are naturally pushing data to the application. To help handle this, Redland wraps these callbacks and turns the data push into a *Stream* pull.

6. Implementation

This section describes the detail of implementing the classes in Redland including the storage classes and the ways used to help make Redland work better with applications.

6.1. Model and storage

The *Model* class is the main application interface for Redland as shown in Fig. 3. Despite this, most of the functionality of the class is provided by other classes. *Storage* deals with all statement-centric and node-centric queries of the model and *Query* handles the other query syntaxes. This makes the *Model* class rather light but it is the key interface for the application, and it is here where functionality for model layering is provided and convenience methods can be easily added.

The *Storage* class implements managing the storage modules via a factory and also handles

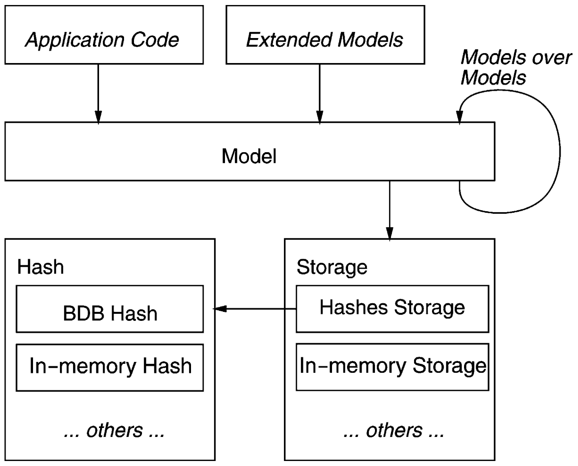

Fig. 3. Redland model layers.

optional parts of the storage module API by converting between internal interfaces. From the application point of view, this is never visible although some operations on the *Model* may be observed to be slower. For example, a storage module may implement only the required statement query methods and not the node based query methods in which case *Storage* will translate between these.

6.2. Storage modules

The main storage implementation for Redland is using multiple *Hash*es to store the statements. This is based on previous work done by the Mozilla project and Guha [15], as well as internal projects at ILRT. A *Hash* in Redland is a map of a key to a value with duplicates allowed. In this implementation, *statements* are stored using three *Hashes*, whether in memory or using a persistent hash (such as BDB) as described in Table 2.

The hashes are used both for the statement queries and the node centric ones. The former are provided by serialising the hash and filtering via the querying statement. This can be very slow for large models so the node-centric indexes are used when only one of the elements of the statement is blank. Node-centric queries mean querying using the *Model* relative to a particular resource node or arc. The SP2O hash finds outgoing nodes from a

Table 2
Multiple hash storage

Hash	Key	Value	Optimises node-centric query
SP2O	Subject and predicate	Object	Get targets of (source, arc)
PO2S	Predicate and object	Subject	Get sources of (arc, target)
SO2P	Subject and object	Predicate	Get arcs of (source, target)

resource with a given arc, the PO2S hash finds incoming nodes with a given arc and destination and the SO2P hash finds the arcs between two given nodes. These combinations of indexes have been found to be quite useful in experiments and testbeds implemented previously at ILRT, without the need to have full combination of indexes.

There are other potentially useful hashes that might be maintained including incoming and outgoing arcs indexed for particular nodes. These choices might be suitable for an option on the storage hash or for user configuration of which statement parts are indexed. The current hash storage module has hooks for such a facility but no current interface to it.

In the future it may be that application-specific indexes will be added to the hashes for optimising queries or properties that are used a lot. The rdf:type property is one that can be refered to often in applications that ask a lot of schema or RDF typing queries and could be worth optimising for. It could either be done at the storage level or higher up where the type system calls might be intercepted by the model and handled in a different way more appropriate for type hierarchies and detecting loops. This is an example of where the flexibility of the application framework can provide different ways to handle application requirements, without making changes to any application interface.

6.3. Statements and nodes

Statements contain three *Node* objects representing the different parts of the RDF statement and these *Nodes* have two main types—resources which have URIs and Literals. In Redland, literals include the string content, the xml:lang and xml: space properties and whether the content is XML content (as declared by the RDF parseType Literal attribute). RDF statements are RDF resources in the RDF model and in Redland *statements* can be used wherever a resource *Node* can be used.

Node objects are used many times inside the applications to represent resources with URIs and thus need to be handled efficiently so that it is easy and quick to create, destroy and manipulate them. For this purpose, an internal node factory is used to ensure that references to a node with a particular *URI* are shared, using simple reference counting (a similar factory is also used for the *URI* class). The RDF model and schema pre-defines concepts such as rdf: type which are used internally but are also often used in application code. These concept resources are pre-defined in Redland so that applications can easily refer to these dynamically-created nodes in compiled code. This also makes it easier for Redland or modules to check and optimise for special use of RDF internal concepts, such as typing, where additional functionality might be wanted—for example, checking that there are no loops in the type tree.

6.4. Parsers and serialisers

The *RDF model and syntax specification* defines a syntax for RDF in XML and this needs to be parsed in order to create or add content to models. The *Parser* class provides access to parsers for this syntax. When development on Redland was begun there were parsers in C, Java, Prolog and other languages, but it was not clear how they compared. Modules were written to wrap the Java parser and call the C one plus allow room for more parsers to be added later if necessary. The interface that these external parsers offer is usually a triple of subject, predicate and object with a heuristic having to be used to guess(!) the type of the object—literal or string—in some cases. This has improved since with the addition of a newer C parser called Repat by Jason Diamond which has a better interface.

The XML syntax is just one potential source of RDF models from a syntax or encoding format and in order to handle that, this class allows modules to register themselves as handling particular MIME types or handling a syntax conforming to a URI. This flexibility means that, for example, modules could be added that extracted or synthesised RDF metadata from image formats such as PNG and JPEG or interpreted MP3 ID3 tags as RDF properties.

Serialisers take a model and emit syntax, either for the purposes of creating a stand alone document representing a *Model* for for other purposes such as delivering as a service. For example, there could be an HTML serialisation that used some policy to flatten the RDF graph into a tree rendered as lists.

6.5. URI and WWW

These modules abstract URIs and provide ways to resolve them via HTTP or other requests. These requests are handled by the *WWW* module that provides a simple interface and a way to return the results. This module is one that is likely to be replaced when Redland is embedded inside an existing Web application since it will already have probably better functionality to do this. It also would be a big problem if Redland blocked an entire application while it waited for I/O from the Web.

6.6. Target language interfaces

The Perl and Python language interfaces were both written using the same interface generation tool called SWIG [16]. It takes a definition of the Redland C interface and automatically generates equivalent functions in the target language via some glue code. These simple functions were then used to create classes in the target languages, calling the Redland C functions inside the methods to perform the actual methods and class operations. The target language classes directly paralleled the Redland classes, with slight changes to accommodate target language metaphors, interfaces and types.

6.7. Features

Application-level access was provided to options, alternative implementations and functionality checking of modules with the features concept. A feature is a key: value pair that can be queried or set for the modules, where the keys are URIs describing the feature. This is modelled after the Java Properties class and a similar technique on the SAX XML parser. An example of where this is used is in the parser modules to indicate or set whether a parser supports the aboutEach and aboutEachprefix attributes, which are not commonly supported.

6.8. Configuration, building and installation management

Redland uses the GNU automake and autoconf tool suite to handle the complex configuration needs of providing a portable system that is easy to use despite having many modular parts. The tool suite tests for features on the operating system that it is installed on and can then include them at configure time if they are present. autoconf also provides user configuration control by options, and these are processed by Redland to select the modules, BDB installations, XML and RDF parsers and other features that are required. The tools finally handle compiling and installing Redland into the standard places for C header files, libraries and documentation.

On Linux, further support is provided by Redland to create RPM packages of Redland that can be installed by users or developers without the need to compile it. This can be automated by package management tools so Linux tools that depend on Redland could install it without any user involvement.

6.9. Infrastructure

Redland also includes some infrastructural support such as a debugging memory allocation tracker that can be removed from application code and linked with the application's own memory management routines; error and warning handlers that can be customised by the application; functions for manipulating temporary files and simple parsers for configuration strings used as parameters for some constructor and class initialisation calls. The latter could have used a *Hash* object but that might be impossible when configuring Redland before the *Hash* class has been initialised.

6.10. Consequences of implementing in C

The use of C for the reasons given above in Section 4—interfacing, performance, portability does have several downsides. The resulting code ends up being rather low level and tricky since all memory allocation and string handling has to be done by hand. If Redland had been implemented in C++ it could have taken advantage of C++'s built-in support for the above plus objects, interfaces, templates and libraries. However, it would have been more difficult to embed in pure C applications, since C calling C++ is rather unusual.

There are also some related technologies such as the OMG interface description language that could have been used to define the interface to Redland but at the time of starting this project (June 2000) the CORBA language mapping had no support for Perl which was a key implementation requirement. This would have limited the deployment languages to those supported, which also excluded some other potential targets such as non-OO Tcl.

However, there is a way around this due to the design of Redland and the heritage of C++. Redland was written in C but implemented in a C++-compatible way and it can be compiled as C++. Thus it is easy to add a set of thin C++ wrapper classes to provide a C++ interface—similar to the way that SWIG adds wrapper classes for the scripting interfaces. This would create a C++ interface that could then be used for providing CORBA object brokers or brokers for similar systems.

7. Redland applications

Redland is a maturing system and the majority of the core work has been implemented and extensively tested. It has now reached the stage of being stable, reliable and and has had several public releases. Two applications have been written by the author to demonstrate the code in use on the Web.

7.1. RDF demonstration of model with persistent store

Redland was used via the Perl language interface to provide a Web-accessible demonstration of using Redland to work with the RDF model, allowing users to submit RDF content to a persistent store, make queries, follow the results and try different parsers that Redland supported—five at present. These parsers showed differences in what RDF/XML they handle which was useful feedback into issues of syntax and parsing.

The database was tested with a copy of the open directory data dumped as 'RDF' [17] (after cleaning syntax mistakes) and 1.3M statements were stored using BDB hashes on disk. This was not the full data set since the data dump still contained syntax errors (not well formed XML) around 2M lines into the 12M lines of output. The resulting RDF statements could be returned from a query at a rate of ≈1800 statements/s from a relatively busy disk.

7.2. RSS 1.0 demonstration

RSS is RDF Site Summary and is a lightweight metadata format that allows content to be simply described primarily for syndication, aggregration and other purposes such as building portals. The RSS 1.0 [18] specification takes an older version of the format and re-introduces it as an RDF application (in earlier versions it was RDF). The Redland demonstration uses the RDF model as the RSS model and defines some application specific methods that the RSS model concepts may have. It was a simple and quick job to add a wrapper Perl class that read the RSS content (in RDF/XML format) into a model and provide convenience interfaces for the RSS concepts. The resulting model was then rendered in a simple HTML output format.

7.3. WSE demonstration

The Web search environments project funded by ILRT uses Redland to provide RDF interfaces over three systems, with speed and flexibility. A non-RDF database of over 360,000 Web pages from a Web crawler is linked with an RDF database of 32,000 structured records and a third RDF database of relationships between the Web pages. The resulting system stores 2.5M RDF statements using nearly 2.3 Gb of BDB hashes is quick to search and can answer over 6000 triple queries/s.

8. Conclusions and future work

The RDF open directory test processes 100 Mb of RDF data using 3 Mb of memory while running, and finished with no memory loss. This was due to the extreme care taken while writing Redland to make sure that no resources were lost. This ensures it can run in long-running processes such as Web services or daemons and be a good neighbour. Redland runs in a small amount of memory because during configuration it dynamically links to the maximum it can with existing system modules that provide digests, hashes (BDB), XML parsers (such as libxml, expat) so that the total memory (code and data) used by the Redland is minimised.

The RSS and RDF Web demonstrations show that Redland provides a high-level interface to RDF that can allows the easy creation of RDF Web services. These could easily be rewritten to provide some Web-based services such as XML-RPC [19] or SOAP [20] using tools that are emerging for those protocols or from a C++ interface that could be created using the method described in Section 6.10. The multiple language interfaces has proved very flexible, with help from SWIG, giving the same interface across five languages quickly and relatively easily.

The compile and install out-of-the-box provided by the automake and autoconf tools configuration makes building and installing Redland a three-line job for most systems and this has been confirmed on five major Unix/POSIX architectures with different compilers, word lengths and endianness demonstrate Redland is mature and portable.

The development of Redland continues to complete and extend the functionality described here plus new developments on mechanisms for grouping statements and experimentation with extracting RDF from other formats such as XHTML and from metadatax embedded in images. The author has also written a new RDF XML-syntax parser which works better with Redland and handles the NTriple format.

Redland implements a powerful modular, object-based library for manipulating the RDF Model and parts—statements, resources and literals. It provides equivalent and full APIs in C, Python, Perl, Tcl and Java. Redland contains modules for multiple parsers for reading RDF/XML and other syntaxes, storage for the models in memory and persistently and flexibility to extend or modify it using layering, modules and factories.

Redland is free software (LGPL/GPL) and open source software (MPL) and available at `http://purl.org/net/redland/` along with links to the demonstrations.

Acknowledgements

Thanks to Nicky Ferguson, Dan Brickley and the Semantic Web Group at the Institute for Learning and Research Technology (ILRT), University of Bristol and Hewlett Packard Labs, Bristol.

References

[1] O. Lassila, R.R. Swick (Eds.), Resource description framework (RDF) model and syntax specification, W3C Recommendation, 22 February 1999, http://www.w3.org/TR/REC-rdf-syntax.

[2] D. Brickley, R.V. Guha (Eds.), Resource description framework (RDF) schema specification 1.0, W3C Candidate Recommendation, 27 March 2000, http://www.w3.org/TR/2000/CR-rdf-schema-20000327/.

[3] D. Brickley et al., Mozilla—resource description framework (RDF), http://www.mozilla.org/rdf/doc/.

[4] J. Saarela, S. Melnik, et al., SiRPAC—simple RDF parser & compiler, http://www.w3.org/RDF/Implementations/SiRPAC/.

[5] S. Melnik, Generic interoperability framework (GINF) project, Digital Libraries Project, Database Group, Stanford University, http://www-diglib.stanford.edu/diglib/ginf/.

[6] R. Daniel, RADIX a proposal for an RDF API, posting to WWW-rdf-interest list, December 1999, http://www.mailbase.ac.uk/lists/rdf-dev/1999-06/0002.html.

[7] P. Hannappel, Summary of recent discussions about an application programming interface for RDF, University of Essen, Germany, April 2000, http://nestroy.wi-inf.uni-essen.de/rdf/sum_rdf_api/.
[8] D. Veillard, rpm2html: a generator of Web pages for RPM package, http://rpmfind.net/linux/rpm2html/.
[9] D. Beckett, Deploying RDF in a large scale mirror service, WWW9 Developer's Day Semantic Web Track, http://purl.org/net/dajobe/talks/www9/.
[10] E. Miller, An introduction to the resource description framework, Dlib Magazine, May 1998, ISSN 1082-9873, http://www.dlib.org/dlib/may98/miller/05miller.html.
[11] S. Melnik, Storing RDF in a relational database, http://www-db.stanford.edu/~melnik/rdf/db.html.
[12] T. Berners-Lee, Building the future, slide in XML and the Web, XML World 2000, Boston, 6 September 2000, http://www.w3.org/2000/Talks/0906-xmlWeb-tbl/slide9-6.html.
[13] S. Melnik, An API for RDF, 2000, http://www-db.stanford.edu/~melnik/rdf/api.html.
[14] D. Brickley, L. Miller, RDF, SQL and the semantic web—a case study, ILRT, October 2000, http://www.ilrt.bris.ac.uk/discovery/2000/10/swsql/.
[15] R.V. Guha, RDFDB—an RDF database, http://www.guha.com/rdfdb/.
[16] D.M. Beazley, SWIG: an easy to use tool for integrating scripting languages with C and C++, 4th Annual Tcl/Tk Workshop, Monterey, CA, 6–10 July 1996, http://www.swig.org/.
[17] Open directory RDF dump, http://directory.mozilla.org/rdf.html.
[18] R. Dornfest (Ed.), RSS 1.0 specification, 19 December 2000, http://purl.org/rss/1.0/.
[19] D. Winer, XML-RPC specification, 23 November 1999, http://www.xmlrpc.org/spec.
[20] D. Box et al., Simple object access protocol (SOAP) 1.1, W3C Note, 8 May 2000, http://www.w3.org/TR/SOAP/.

David Beckett is a technical researcher who has been developing, researching and deploying Internet research discovery systems and metadata since 1993. He is a long-time member of the Dublin Core Metadata Initiative, developing the Dublin Core Metadata Element Set (DCMES) and is the editor of the first DCMES in RDF/XML document. David has been using with RDF since 1998 and has deployed several services with it and DC. He joined ILRT in June 2000 and has been integrating RDF, web crawling and human-cataloguing of the web as part of the Web Search Environments project and for this developed the Redland RDF system. David participates in RDF standardisation as a member of the W3C RDF Core Working Group and has co-edited the first two W3C Working Drafts produced. He has also given several invited presentations at international workshops and conferences on RDF, Dublin Core and the Semantic Web.