An RDF Framework for Resource Discovery

Franklin Reynolds
Nokia Research Center
5 Wayside Road
Burlington, MA, 01803 USA
1-781-993-3619
franklin.reynolds@nokia.com

ABSTRACT
Resource discovery is a problem common to almost all distributed systems. Instead of resulting in a one or a small number of discovery mechanisms, completely different and incompatible discovery services have proliferated. It can be reasonably argued that a discovery protocol optimised for small wireless LANs, such as Bluetooth piconets, is unlikely to be suitable for enterprise scale networks or the Web. Drawing inspiration from the database community's success with standardising APIs, data models and query languages, we propose a protocol independent framework based on RDF. The framework consists of a flexible data model, metadata API and query language.

General Terms
Algorithms, Management.

Keywords
RDF, Discovery Protocols, SLP.

1. INTRODUCTION
The need to resolve names and discover resources is common to almost all distributed systems. Problems related to discovery occur in large networks and small, in dynamic networks and relatively static networks and at high levels and low levels of network abstraction. Many different, successful discovery protocols and applications have been developed over the years including:

- Dynamic Host Configuration Protocol [DHCP]
- Bluetooth Service Discovery Protocol [SDP]
- Universal Plug and Play Simple Service Discovery Protocol [SSDP]
- Service Location Protocol [SLP]
- Ligtweight Directory Application Protocol [LDAP]
- Web search engines such as [Google] and [AltaVista]

These examples are intended to illustrate the range of discovery protocols in current use. DHCP is a very low level service used primarily to configure the basic network services of a host before it can communicate with other hosts on the Internet. No one would seriously consider DHCP a suitable basis for a Web search engine. Different protocols are needed for different applications. And yet, while each discovery protocol is different, the fundamental problem solved by each service is similar.

Given this similarity, it is not surprising that study of different discovery services reveals common issues, design patterns or components shared by most if not all of the servicess. We have found it useful to characterize discovery services by their approach to the following issues:

1. Discovery of the discovery service (for example, the use of DHCP to discover the SLP service)
2. Advertizement (for example, SSDP advertizes services via multicast to clients and SLP DAs advertize themselves so that SAs can register their services and clients will know to use new DAs)
3. Existence of a Registry or Centralized Name Server(s) and how they are organised
4. Query Protocol (multicast or unicast)
5. Query Language
6. Metadata model, vocabulary and schemas

None of these services interoperate. The metadata used to describe resource in one discovery service must be duplicated in an incompatible format to be used by a different discovery service. There is no common interface applications can reuse when using two or more of these services. In some cases there are strong similarities, such as the query languages of SLP and LDAP. But interoperability has never been a compelling goal.

One of the strongest lessons learned by the database community over the years is the importance of application level interoperability. A common data model, such as the relational model and a common query language, such as SQL, provides considerable interoperability for applications that use SQL. Among other benefits it allows UI and application level innovation to proceed independent of specific database products and it allows the database vendors to innovate without the risk of breaking all existing applications.

Theres are strong similarities between simple databases and service discovery applications and yet the service discovery community has not learned to provide application level interoperability.

2. Application level interoperability
We propose the use of a single, flexible metadata model, metadata language and query language to serve as the basis for interoperability between service discovery systems. This is similar in spirit to the database community choosing a relational data model and SQL for interoperability. While it may be that no single choice would reasonably support all possible discovery

services, we are confident that we can cope with most important services.

We choose the W3C's Resource Decription Framework [RDF] to serve as the basis for our framework. RDF was designed as a general purpose knowledge representation language and has a very flexible data model which is more than expressive enough for our needs. RDF vocabularies can be named via XML namespaces and defined according the RDF Schema specification. This permits the definition of application specific vocabularies without the need for a standardization organization, such as IANA.

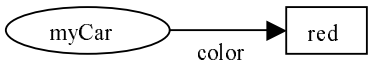
Though the specification for RDF was completed in early 1999, RDF and RDF tools have not matured as we expected. No standard RDF APIs or tools exist that are comparable to DOM [DOM] for XML or pattern matching grammer such as XSLT [XSLT] patterns for XML. Consequently we developed our own RDF API and pattern matching language.

2.1 The RDF Data Model

The RDF data model is a labelled, directed graph. A graph consists of nodes and arcs. Each node and each arc have labels. Labels associated with nodes must be unique within a graph. Labels associated with arcs need not be unique. Each arc in the graph can be described triple consisting of three fundamental RDF entities:

- Resource - a resource corresponds to a node on a graph
- Property - a property corresponds to an arc
- Value - A property can be either a string literal or a Resource

The most simple RDF graph consists of a single node, property, value triple. For example, a triple consisting of a resource named "myCar", a property named "color" and a value of "red" would look like the following graph:



RDF graphs can be represented using XML. A simple representation of the above graph using XML would be:

```
<rdf:Description about=myCar>
     <fdr:color>red</fdr:color>
</rdf:Description>
```

In addition to the graph-based data model, an RDF class system has been defined [RDFschema]. Together, RDF and RDF Schemas provide very powerful and sophisticated tools for modeling information. For our initial work we have choosen to restrict ourselves to the core data model rather than integrate the use of RDF Schemas and possible complications of a federated class hierarchy. Searchable metadata is expected to be encoded using RDF without schemas and the query language is designed for labelled, directed graphs.

Though RDF can be serialized using XML, the RDF data model is actually quite different from the XML data model. The RDF data model is a labeled, directed graph and XML's data model is a tree with different types of nodes. The RDF data model could be expressed using a different syntax but the use of XML has a variety of advantages, including the promise of XML tool reuse. However, DOM, the XML document object model and API [DOM], is not really adequate for RDF.

2.2 A Simple RDF API

Several RDF toolkits have been developed including [GINF], [Jena] and [Redland] but at the time of this work, no standard, official or de facto, has emerged. Proposals have ranged from simple APIs that mimic the data model to much more ambitious efforts that allow for the manipulation of the graph, inference engines and other features beyond the basic data model. For our purposes, a simple approach is sufficient. Our API is a small collection of classes which, define a graph, a node in the graph which has arcs to other nodes and an RDF triple. Once a graph is created, it can be navigated or destroyed but it cannot be changed.

This API is more similar in spirit to DOM than the richer APIs provided by other toolkits. Obvious improvements would be to allow for addition, change and deletion of nodes and arcs from the graph and explicit support for RDF schemas. A general purpose RDF API would need similar features but a simple DOM-like API meets our needs. Our API is based on Java and while still under development, it has already proven to be a useful tool.

The class definitions for the API are:

```
public class RDFGraph {
  public RDFGraph( String filename ) ;
  public Enumeration elements() ;
  public Enumeration elements(RDFNode
root);
}
public class RDFNode {
  public Enumeration elements() ;
  public void display() ;
}
class Triple {
  public RDFNode resource ;
  public String property ;
  public RDFNode value ;
  public boolean stop_hint ;
}
class Depth_first implements Enumeration {
  public boolean hasMoreElements() ;
  public Object nextElement() ;
  Depth_first( Enumeration enum, Stack s )
;
}
```

The class RDFGraph is intended to contain a graph described by an RDF fragment. RDFGraph provides two views of the graph.

One is the list of all the triples in the graph and the other is a depth-first spanning tree of the graph. RDFGraph has a constructor that takes a file name as an input parameter. The file should contain the list of RDF triples to be used to construct a graph. The SiRPAC [SIRPAC] RDF parser is used to generate the file of triples. Though unusual, the list of triples may describe a disconnected graph or multiple graphs. To deal with this, an RDFGraph object may actually be a collection of graphs. In the future, a new collection object may be added to the API to explicitly provide for aggragating graphs.

An RDFNode object is equivalent to a resource. An RDFNode object is a node in a graph and the object contains the list of triples that describe arcs eminating from the node. Each RDFNode contains a Resource, all its Properties and the Values of each Property. An Enumeration method exports the list of Properties. The Values of Properties are other RDFNodes. An RDFNode with no Properties corresponds to a Value, as opposed to a Resource. Finally, in a glaring violation of good object oriented design, RDFNode exports its member data for easy manipulation by other objects rather than exporting methods that perform the manipulation.

The data model for RDF allows a single graph to contain merged branches, cycles and multiple roots. To simplify the task of navigating within a graph, Enumeration methods are provided as a means of traversing all the arcs of all the graphs associated with the RDFGraph object. When an RDFGraph object is constructed, a depth-first spanning tree is overlaid upon the graph. The "stop-hint" member of a Triple is used to detect leaves of the spanning tree. The leaves terminate cycles and merged branches in the graph. Graph traversal using the spanning tree results in visiting every arc once and only once. The Depth_first class, which implements the Enumeration abstract class, works in conjunction with the enumeration methods of RDFNode to provide the mechanisms for walking the graph.

The following is an example of code that walks the graph and displays each RDFNode:

```
// perform a depth first walk of
// the graph and display each
// node as it is traversed.

Enumeration e = graph.elements();
while (e.hasMoreElements()) {
  ((RDFNode)e.nextElement()).display();
}
```

2.3 Directed Graph Query Language (DGQL)

There is active work to define XML based query mechanisms at the W3C and other organizations. Though several different research groups have considered various approaches to RDF based queries, no standards have emerged. DGQL is our attempt to define a simple query language based on graph matching.

Object-oriented database query services usually provide schema or class based query mechanisms. The schemas or object definitions provide the basis for dealing with structured data. RDF provides a schema framework that may be suitable for a schema based query languague, but the use of RDF schemas is optional and adds more intellectual complexity than we desire. Consequently DGQL does not use schemas or class definitions but it can be used to query graphs created using RDF Schemas. The use of RDF Schemas, specifically inheritence, raises some interesting challenges associated with indefinitely recurring patterns that we hope to explore in the future.

Our goal was to design a simple, general purpose, query language suitable for automated queries and upon which other languages could be built. We expect that as the Semantic Web evolves, a wide range of query languagues and services will be built and deployed. These may include new, natural language based services for human interactions, domain specific, schema based, search engines and support for legacy query services. Performance optimizations are not a primary goal, but we are sensitive to the need for a query service to provide reasonable performance. DGQL is designed to hide some of the most peculiar characteristics of RDF in order to serve as a general purpose, though not necessarily human friendly, tool that simplies the task of providing higher level query services. DGQL is a pattern matching language. Both Resources (Nodes) and Properties (Arcs) can be tested. The syntax is based on simple, parenthesized S-expressions. The AND and OR logical operations are supported. The S-expressions can be nested which is how the direction of an arc (Property) is expressed.

2.3.1 Syntax

Basic DGQL statements have the general form of:

```
( OpCode  Parameter List )
```

Legal OpCodes are NTEST, ATEST, AND and OR. NTEST is the command to match a pattern to a node and ATEST is the command to match a pattern to an arc.

A Node (Resource) test has the following syntax:

```
(NTEST pattern [arc test|logical])
```

The [...|...] notation is used to indicate that the last parameter is optional. If it occurs, it can be an arc test or a logical command.

An Arc (Property) test has the following syntax:

```
(ATEST pattern [node test|logical])
```

The [...|...] notation indicates that the last parameter is optional. If it occurs, it can be an node test or a logical command.

The pattern used in node and arc tests can be a string or *. Strings are tested for equality. The * character matches anything. Future work may include the introduction of regular expression string matching rules.

Node tests cannot be parameters to a node test and arc tests cannot beparameters for arc tests. This is because the nesting of

tests is how the relationship between nodes and arcs is expressed. In a graph, nodes are only connected to other nodes via arcs. Similarly, arcs are associated with nodes, not other arcs.

A logical command has the following syntax:

```
(AND|OR list_of_tests)
```

Logical AND and OR commands take a list of tests as parameters. Because of the nesting rules for nodes and arcs, a logical command that is a parameter of a node test can only have arc tests in its parameter list and a logical command that is a parameter of an arc test can only have node tests in its parameter list.

2.3.2 DGQL Class

DGQL queries have been implemented in Java as the DGQuery class. To fully process a query against an RDFGraph, the query must be applied to each node in the graph. To do this, the DGQuery object uses the depth-first graph traversal methods exported by RDFGraph and RDFNode.

As the query is applied to a node the query processing may traverse some fraction of the graph. Currently DGQL does not provide recursive queries, but this is planned for the future. To avoid infinite loops that could occur with the combination of recursive queries and cyclic graphs, the query engine will use the lstop-hints of the RDFGraph spanning tree.

DGQuery exports a constructor that takes a String containing a valid query as an input parameter. DGQuery provides an evalq() method that applies the query against an RDFNode supplied as an input parameter. As the constraints in the query are evaluated, evalq() may traverse any reachable portion of the graph.

```
public class DGQuery {
    public DGQuery( String query );
    public boolean evalq( RDFNode
current_node,
                          StringBuffer ms)
        throws bailout_exception ;
}
```

An example of how to use the DGQuery class follows:

```
// input a stream of queries and
// look for a match

try {
   // Open the queries input file
   BufferedReader input = new
     BufferedReader( new InputStreamReader(
     new FileInputStream(args[3])));

   // read next query from file
   // EOF == null
   while ( null != (query_line =
           input.readLine()) ) {
     StringBuffer match_string =
                  new StringBuffer();
     DGQuery q = new DGQuery(query_line);
     e = graph.elements();

     while (e.hasMoreElements()) {

       if (q.evalq(RDFNode)e.nextElement(),
                        match_string ) ) {
           System.out.println(" success! "
+

match_string);
           break;
       }
     }
   }
 } catch( IOException e1 ) {
     System.out.print( "Error: " + e );
     System.exit( 1 );
 } catch (bailout_exception ee) {
     System.out.println("parsing error ");
 }
```

3. Query Results

The reply from a query includes all the matching sub-graphs. Consider a very simple graph that models a collection of four objects, including the name and URL of each object:

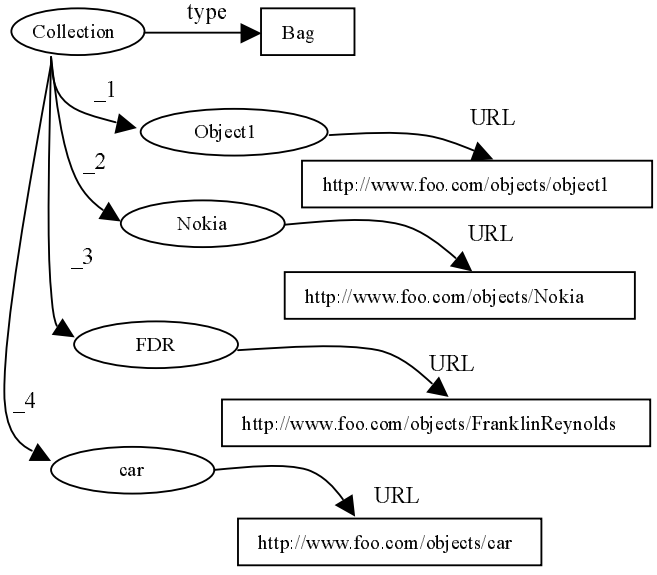


Figure 1.

If the query's constraints can be satisfied, then the matching graph(s) are returned. The format of the reply is somewhat similar to the query syntax

```
(N resource
  (P list_of_props (N list_of_resources ))
  (P list_of_props(N list_of_resources)))
```

Between wildcards and OR conditions it is possible for many patterns to match a single query. All instances of matching patterns can be returned. Though unusual, multiple instances of property with the same name but different values can be associated with a single resource. Thus, even an exact match rule can match multiple properties. If there is more than one match for an individula rule, a comma separated list of each instance of a matching pattern can returned. In some cases, this list can be quite large.

There is an irregularity in the way the query reply is constructed. DGQuery.evalq() takes two parameters: an initial node in the graph and the reply string. The first rule of the query is only applied to the initial node. As shown in the example, to apply the query to all nodes in the graph requires the use of the RDFGraph walking methods. This means that if the first rule is a wild card that matches every thing, then the maximum number of replies will equal the number of nodes, rather than a single reply with a list of nodes matching the inital rule. Though this may result in many replies it is still on the order of N where N is the number of nodes in the graph.

The reason for the irregularity is to permit the selection to be short circuited if application is only interested in the first successful match. For example, you may wish to apply a query to a particular node, instead of the entire graph. This mechanism could be used to traverse the graph using a series of queries.

4. Experience and Lessons Learned

4.1 Prototype Discovery Services

In an effort to experimentally validate the idea that a common metadata toolkit and query language could provide some level of interoperability, we implemented DGQL extentions to SLP and a web based discovery service. The protocols, usage models and user interface to each service is quite different.

The Web search engine has a synchronous, request-response protocol based on HTTP. The Web and Web search engines were designed to be used interactively by human beings. The User interface is based on HTML Forms. Our web search engine is intended to be reminiscent of other search engines. While it is certainly possible for a computer program to construct the proper HTTP messages to interact with the Web discovery server, this is not the typical usage case.

The design of SLP has less emphasis on interactive human users. SLP uses both synchronous and asynchronous request-response protocols. The synchronous SLP protocol is similar in behavior to the Web search engine protocol. A single request results in a single reply, though the reply may contain multiple matches. Asynchronous SLP requests can result in multiple replies. Any of those replies could contain multiple matches.

The RDF APIs, DGQL and the Java classes that implement DGQL have been used to implement new, structured queries as additional funtionality to SLP and a prototype Web search engine. The reuse of these components resulted in several benefits:

- The use of RDF as the data model makes it possible the use of structured, almost arbitrarily complex metadata and multiple, user defined vocabularies. This allowed us to model the information we wanted to advertise in a natural fashion, rather than being forced to flatten or otherwise coerce the metadata into some simple format, such as the schemas used by SLP.
- The same metadata can be used and queried using both protocols. Most discovery services use completely different data models and vocabularies. In order to advertise a resource to SLP clients and Web clients requires the resource to be described in completely different languages. There is no guarantee that it is possible to create equivalent descriptions in different languages. We were able to use the identical RDF description files for both the SLP and Web based discovery services. This makes it easier to reduce undesired information inconsistencies between services.
- The availability of the DGQL classes greatly simiplified the task of reimplementing DGQL functionality for multiple discovery services. This was possible because our prototypes were all implemented in Java. If we were concerned about optimising performance we might need to reimplement DGQL classes in a different language such as C.
- The common query language made it possible to send the same queries via different protocols. Except for differences due to the behavior of their protocols, our SLP and web discovery services provide consistent search behavior for applications and users.
- This approach was surprisingly efficient. SLP and some other discovery protocols can operate in a peer-to-peer mode. Clients multicast their queries to all hosts, rather than directing their queries to a single server. This is particularly useful in ad hoc networks that are composed of many different types of hosts, including very small devices and services, but have no centralled administered network infrastructure (such as name servers, etc.). Since DGQL can act on the RDF data model rather than the RDF language, small clients can have very compact representations of the metadata and the parsing of DGQL requires only a few hundred lines of Java. This eliminates the need for XML and RDF parsers on these resource limited devices.

While the use of a protocol independent query language has advantages, it does not fully isolate the user or application from the need to know about the different types of discovery services available. The scope and accuracy of the knowledge of each service as well as the expense associated with using each service, may be quite different. It is not always desirable to send every query to every known discovery service.

Tools such as Apple's Sherlock [Sher1180] and other meta-search mechanisms attempt to provide a framework for aggragating multiple discovery services or selecting the appropriate discovery service or services for each query. In the future, we hope to explore the use of "context-awareness" for the dispatching of

queries to appropriate discovery services. In general, we feel that more work in this area is warranted.

4.2 Some problems

We judged our prototyping experience a success. The basic idea we were interested in exploring was the practicality of a common metadata model, metadata toolkit and query language for different discovery protocols. Our RDF based toolkit allowed us to use very different protocols to advertise and discover the same resources using the same resource descriptions and queries. However, there were a few bumps in the road.

Though RDF was standardised a couple of years ago, there are still no standard or de facto standard RDF APIs or tools. Though we believe our RDF API has a simple elegance, the only reason we invented an RDF API was because there was no standard API to use.

In the absence of a standard, DOM-like API, a standard triple API for RDF parsers would be valuable. The format of RDF triples is not standardised, and as a consequence, each RDF parser is different and it is quite difficult to write RDF applications that are parser-independent. For example, the treatment of anonymous resources is underspecified, resulting in parser specific behavior.

An RDF triple consists of a resource, a property and a value. The value can be a string or a URL. RDF does not explicitly support other types of values such as integers. This limits DGQL to string matching operations. Numeric comparisons such as ">" and "<" would be useful. One approach would be to add the numeric test operations to DGQL and to use the RDF schema mechanism to define a RDF vocabulary for service discovery. Applications that need to define their own vocabularies would be free to inherit data types from the discovery vocabulary. Unfortunatly, introducing types and the use of RDF Schemas adds a great deal of complexity to the system.

The RDF specification seems to permit multiple instances of identical triples. The benefit of multiple, identical triples is unclear. While it is not particularly difficult to create appropriate algorithms deal with non-unique triples, it was our experience that non-unique triples lead to human errors and misunderstandings.

There is a debate within the RDF community on the best way to deal with collections of RDF graphs. One approach is to maintain each graph as a separate entity and another is to combine all available triples into a single "triple store". We have chosen to maintain the separateness of each graph. This preserves the information of what was and what was not asserted by each graph. One disadvantage of this approach is that it is difficult to compose a query that tests assertions made in multiple graphs.

RDF is intended to allow for the creation of application specific vocabularies. This flexibility allows new applications to describe themselves without being dependent on standard vocabularies but it creates a challenge for users of discovery serivces. Usually, a discovery service includes a standard vocabulary and there is some organisation that controls modifications to that vocabulary. This simplifies the task of advertising resources and constructing queries. In the absence of a priori knowledge of the language used to describe resources, clients will need mechanisms to discover and utilize the languages used by different resources. This problem is not unique to RDF and is an active area of research.

5. Summary

The goal of this project was to investigate the use of RDF as a basis for application level interoperability between discovery services. We developed an experimental RDF toolkit and prototyped a web based discovery service and extentions to SLP based on the RDF toolkit.

Though the prototyping exercise exposed some of the limitations of our approach, the benefits of a discovery protocol independent query language, data model, metadata language and toolkit were clear.

6. Acknowledgements

My colleagues at the Nokia Research Center in Burlington, MA have been very generous with their time and advice during this project. I am particularly grateful to June Tran, Ora Lassila, Mark Adler, Jarkko Hietaniemi and Louis Theran, who have all contributed to the ideas described in this paper. In addition, June implemented the web based discovery service using early and very buggy versions of the tools.

7. REFERENCES

[1] [DHCP] Dynamic Host Configuration Protocol. R. Droms. March 1997, IETF RFC 2131

[2] [SDP] Part E Service Discovery Protocol (SDP), Dale Farnsworth, ed., Bluetooth Specification, Bluetooth Special Interest Group; http://www.bluetooth.com/link/spec/bluetooth_e.pdf

[3] [SSDP] Simple Service Discovery Protocol/1.0 Operating without an Arbiter. Y. Y. Goland, T. Cai, P. Leach, Ye Gu, S. Albright, http://upnp.org/draft_cai_ssdp_v1_03.txt

[4] [SLP] Service Location Protocol. J. Veizades, E. Guttman, C. Perkins, S. Kaplan. June 1997. IETF RFC 2165

[5] [LDAP] Lightweight Directory Access Protocol (v3). M. Wahl, T. Howes, S. Kille. December 1997, IETF 2251

[6] [RDFschema] Resource Description Framework (RDF) Schema Specification 1.0, Brickley, Guha, eds., World Wide Web Consortium andidate Recommendation; http://WWW.W3.ORG/TR/2000/CR-rdf-schema-20000327/

[7] [RDF] Resource Description Framework (RDF) Model and Syntax Specification, O. Lassila, R. R. Swick, eds., World Wide Web Consortium ecommendation; http://www.w3.org/TR/REC-rdf-syntax

[8] [SIRPAC] SiRPAC - Simple RDF Parser and Compiler, Janne Saarela, http://www.w3.org/RDF/Implementations/SiRPAC

[9] [DOM] Document Object Model (DOM) Level 2 Core Specification Version 1.0, A. Le Hors, P. Le Hegaret, L Wood, G. Nicol, J Robie, M Champion, S Byrne. World Wide Web Consortium Recommendation; http://www.w3.org/TR/REC-DOM-Level-2-Core-20001113

[10] [XSLT] XSL Transformations (XSLT) Version 1.0, James Clark, World Wide Web Consortium Recommendation; http://www.w3.org/TR/xslt.html

[11] [GINF] Generic Interoperability Framework, Sergey Melnik, http://www-diglib.stanford.edu/diglib/ginf/

[12] [Redland] Redland RDF Application Framework, Dave Beckett http://www.redland.opensource.ac.uk/docs

[13] [Jena] Jena - A Java API for RDF, Brian McBride; http://www-uk.hpl.hp.com/people/bwm/rdf/jena/index.htm

[14] [Sher1180] Sherlock's Find by Content Library, Apple Tech Note 1180; http://developer.apple.com/technotes/tn/tn1180.html