

The XRI Polyarchy

Steven Churchill, ooTao, Inc.

February, 2007

Draft 3

=steven.churchill

This Article is About

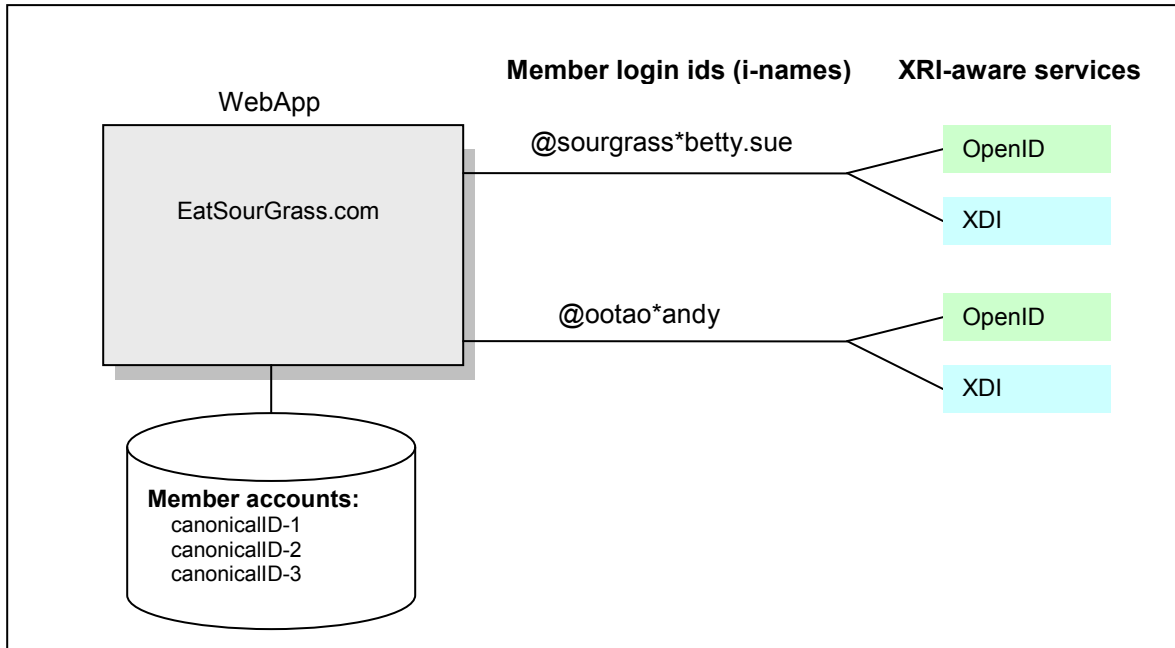
A powerful feature of XRI resolution is its formalization and verification of polyarchical relationships between XRI authorities. This article explores how these relationships are defined, how they are used by applications, and how they are verified by XRI resolution.

Audience

This article assumes familiarity with XRI Resolution, including Authority Resolution, Service Selection, Ref Traversal, and CanonicalID Verification.

Motivation

My website EatSourGrass.com brings together diverse sour grass eaters from around the world in a shared spirit of collaboration and community. Figure 1 shows the site's architecture.



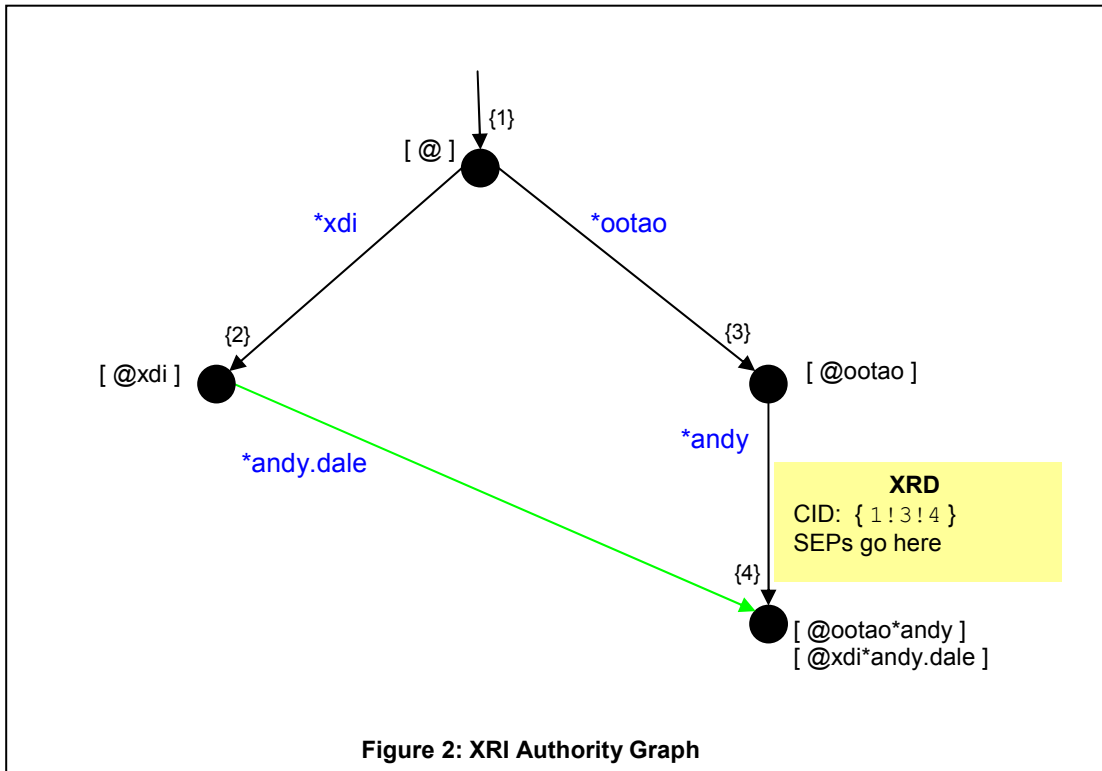
My web application relies upon XRI-aware services used by its members, such as the OpenID authentication service, which it uses for member login, and the XRI Data Interchange (XDI) service, which it uses to share member profile data as well as to manage and share their favorite sour grass recipes.

I personally find this polyarchical XRI authority stuff really cool, so I decided to architect my application so that it uses canonicalIDs (CIDs) as the primary key for application's account database. Because various authority synonyms resolve to the same CID, these synonyms can be used to identify the same member account without the need to register each synonym explicitly.

For example, my Sour Grass application has a member account for the authority @ootao*andy. And because he is also a polyarchical kind of guy, Andy has decided to establish a polyarchical relationship from the authority @xdi*andy.dale to the authority @ootao*andy. The authority XRI (the i-name) @xdi*andy.dale, then, can be considered an *authority XRI synonym* of the authority XRI @ootao*andy.

Why would Andy want to set up this polyarchical relationship? One reason might be that he has previously set up all his profile data in @ootao*andy's XDI service – and he prefers not to do it again under another XDI service for @xdi*andy.dale. Also, he has a fairly secure password (that he can actually remember) for @ootao*andy. He is more than happy for @xdi*andy.dale to simply share @ootao*andy's OpenID authentication service and password.

Figure 2 illustrates the polyarchical relationship.



The graph in figure 2 shows four XRI authorities nodes. The edges are labeled in blue with local authority sub-segment names (such as *ootao). Each edge can have multiple synonyms for these local names (such as *ootao.inc). For illustrative purposes, the nodes are labeled with [bracketed] authority XRIs that resolve to the given node. For example, the two authority XRI synonyms @ootao*andy and @xdi*andy.dale will both resolve to the node in the lower right corner.

The edges are also labeled with local i-numbers (for example {3}). The box in yellow represents the XRD metadata produced by a node's authority resolution service for the given child. For example, the authority resolution service for [@ootao] will return an XRD containing the canonicalID { 1!3!4 } when asked to resolve *andy. The XRD also contains the SEPs (service endpoints) for the authentication service, XDI service, etc. The SEPs are not shown in the diagram.

Finally, the green edge represents a polyarchical parent-child relationship from the node resolved by @xdi to the node resolved by @ootao*andy. With the green line, that same node can now be resolved using @xdi*andy.dale.

As long as resolving @xdi*andy.dale also produces an XRD with the canonicalID { 1!3!4 } then by using this canonicalID as the primary key for my member database, I have designed an application that can treat both @xdi*andy.dale and @ootao*andy as authority synonyms for the same member account. If he wants, Andy can create additional polyarchical relationships to @ootao*andy from his other communities, such as @sourgrass*ad, and so forth. None of these synonyms need to be registered explicitly with my web application. Instead, because they all resolve to the same canonicalID, they can all be used to identify the same member account.

It is important to note that whereas my Sour Grass application is happy to use canonicalIDs and to exploit polyarchical relationships without explicitly registering authority XRI synonyms, other applications may not care about this feature, and, in fact, they may have very good reasons to support authorities that do not provide canonicalIDs at all. One such application may instead wish to use explicit synonyms (such as @ootao*andy) as the member account identifier, and may wish to force other synonyms to be registered explicitly. Another application may want to allow short-lived accounts that allows users to use randomly generated authority identifiers. All these use cases are valid and are supported by XRI resolution.

Securing the polyarchy

I decided to use canonicalIDs for my website's member accounts. When the user comes to my website and enters an authority XRI (i-name), such as @xdi*andy.dale, the first thing my web application does is invoke the XRI resolver with two goals in mind:

- To obtain the canonicalID so that it can look up the member account, and
- To obtain the OpenID authentication service SEP so that it can authenticate the member.

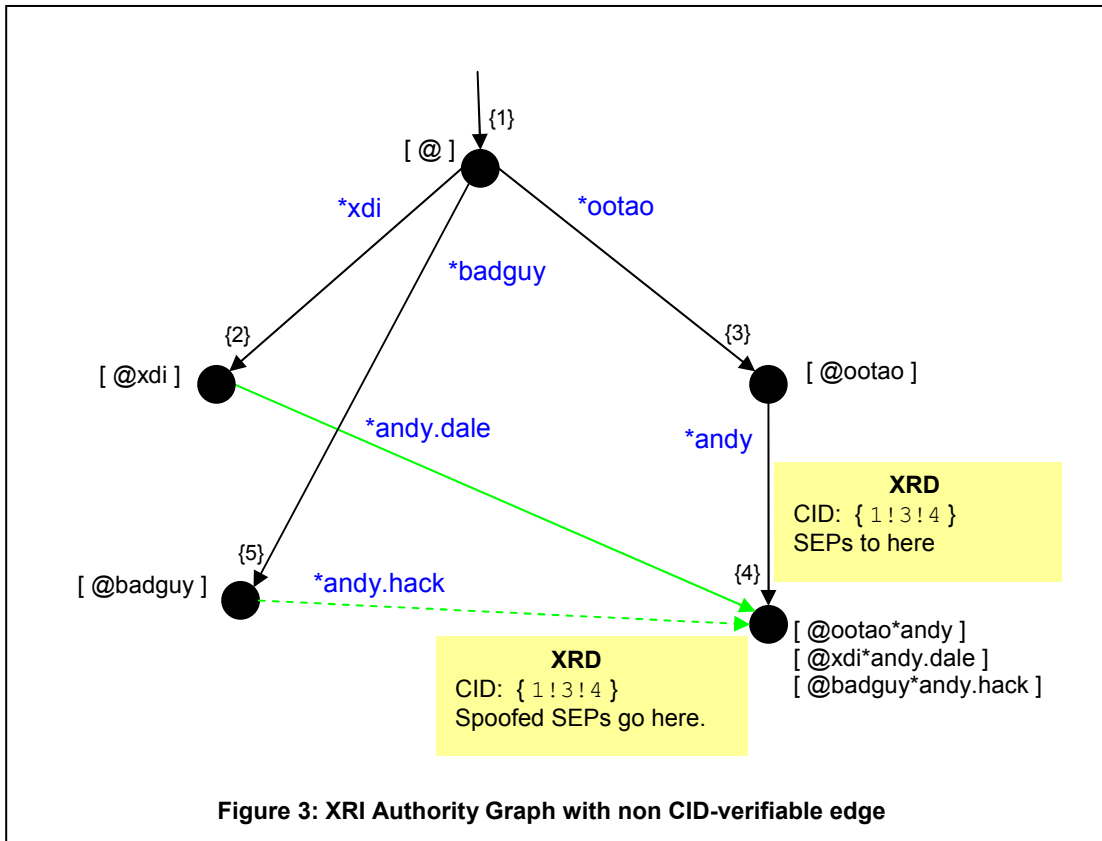
Can you smell a possible spoof here? Here it is: imagine that a bad guy creates an authority named, say, @badguy and that he creates a polyarchical parent-child relationship from @badguy to @ootao*andy as shown in figure 3.

Further say that the bad guy sets things up so that resolving @badguy*andy.hack produces a canonicalID with {1!2!4} (just like the one produced by @xdi*andy.dale). Here's the trick: this XRD contains an OpenID SEP with a URI pointing not to @ootao*andy's OpenID service, but to the bad guy's personal hacker OpenID service. Here's the scenario:

1. The bad guy comes to my website and enters @badguy*andy.hack into the member login field.
2. My webapp invokes the resolver with the resolution media type parameter `_xrd_r="application/xrd+xml"` and the service selection parameter `sep="true"` and gets back an XRD with the OpenID SEP.
3. My webapp extracts the OpenID URI from the SEP and successfully authenticates the member.
4. My webapp extracts the canonicalID { 1!3!4 } from the XRD and looks up @ootao*andy's account from the database.

The spoof is in step 3. Although the XRD has @ootao*andy's canonicalID { 1!3!4 } it contains a spoofed endpoint for the OpenID URI. Rather than this being Andy's OpenID service, it is the bad guy's OpenID service, and it allows him to login with his favorite password. He now has access to Andy's member account. For another nasty trick, the same XRD can contain a spoofed SEP URI for Andy's XDI Service. Now when my webapp goes to get Andy's favorite Sour Grass recipe, the bad guy produces a recipe for pumpkin pie! (The Sour Grass community will not be laughing, I can assure you.)

Figure 3 shows the @badguy authority along with its malicious polyarchical parent-child relationship to authority @ootao*andy. This edge appears as a dashed green line in order to illustrate its malicious nature.



To plug this security hole, the resolver provides a mode that prevents the malicious XRD (shown along the dashed green line in figure 3) from being returned to my webapp. My webapp simply specifies `cid="true"` to the resolver, and this turns on "canonicalID verification".

The following is what is meant by canonicalID verification:

If I resolve an authority XRI for an XRD with `cid="true"`, and a canonicalID is returned in the XRD, then that XRD is guaranteed to be produced by the parent of the canonicalID.

This is saying that the XRD along the edge named `@andy.hack` (the yellow box at the bottom) would never be returned with `cid="true"`. Only the XRD along the edge named `*andy` (the yellow box at the right) will be returned. This is because that XRD is produced by the parent `@ootao { 1!3!4 }` of the canonicalID `{ 1!3!4 }`. If figure 3, resolving both `@ootao*andy` and `@xdi*andy.dale` would return XRDs containing `{ 1!3!4 }` but resolving `@badguy@andy.hack` would not.

Since the resolver will not return to my webapp the malicious XRD, then my webapp is not vulnerable to the bad guy's spoof.

Being able to equate the concept of **canonicalID verification** with **the guarantee that the XRD returned by the resolver has been produced by the parent of the returned canonicalID** is quite important and useful. It is well worth spending a bit of time going over this section trying to cement an understanding.

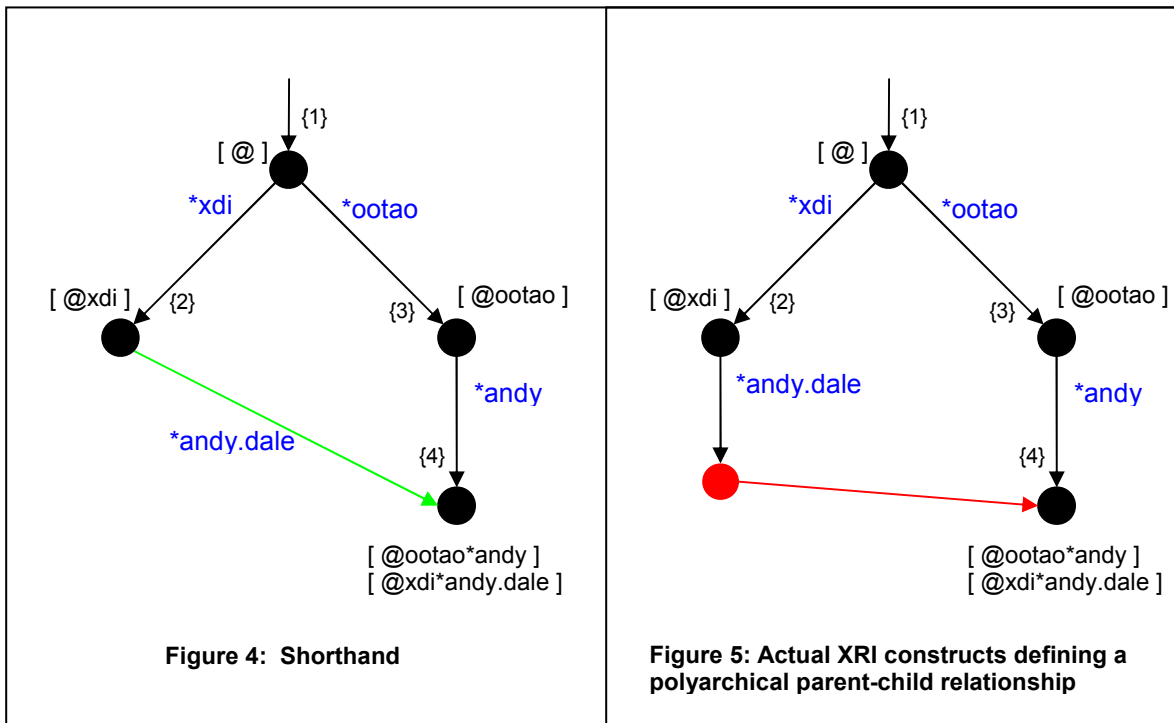
For example, if you understand this concept, then it is easy to see that if two canonicalIDs are returned in an XRD under `cid="true"` then both canonicalIDs must be "synonymous paths" (where a synonymous path is defined as *two paths with the same number of edges that traverse the same nodes in the same order.*)

A reasonable question at this point would be: *how does the resolver distinguish between the good polyarchical parent-child relationship (the solid green edge in figure 3) and the malicious relationship (the dashed green edge)?* The next section answers this by exploring how polyarchical parent-child authority relationships are defined under XRI resolution.

Defining polyarchical parent-child authority relationships under XRI resolution

Perhaps the first thing to say about defining polyarchical parent-child relationships is that XRI resolution provides no *direct* way to define them. The green edge shown in figure 4 (the same green edge shown in figure 2 and figure 3) is really just a shorthand notation for the explicit XRI resolution constructs shown in figure 5.

In other words, in order to form a polyarchical parent-child relationship from the @xdi authority to the @ootao*andy authority, one needs to define a direct hierarchical child under the @xdi authority—this is the node shown in red—and assign to it an xrd:Ref element that targets the child node. This is the red edge.



Let's look more closely at the edges used in the graph diagrams in this article.

A black edge: This is a hierarchical parent-child relationship. (*Hierarchical* here indicates that a given node will have at most a single black edge parent.) This edge and its target node exist when an authority resolution service for a given parent successfully returns an XRD when queried for the name of the edge and a Ref is not followed to obtain the child XRD. For example, if I query @'s authority resolution service for `*xdi` and it successfully returns an XRD (without following a Ref) then both the `*xdi` edge and its target `[@xdi]` node exist in the graph.

A red edge: This is a polyarchical *union* relationship defined using a Ref element in the XRD. We'll explore the meaning of the union relationship below.

A green edge: This is a polyarchical *parent-child* relationship and is really just shorthand for a hierarchical child node and the edge represented by its Ref.

I should clarify that when I say that the black and green edges represent parent-child relationships but that red edges do not, I am speaking about parent-child relationships from an *XRI resolution* standpoint. The red edge is a *directed* edge, and, as such, it has a parent and child from the standpoint of it having a source node on one side of the arrow and a target node on the other. However, from an *XRI resolution* standpoint, the notion of parent-child is all about authority subsegment traversal. The parent appears on the left side of the subsegment delegation character (in our examples the * character) and the child appears on the right. For example, @ootao is the parent of andy in @ootao*andy. Since the red edges are defined by the xrd:Ref element, they represent the *unioning* of the source node with the target node—this is not at all about authority subsegment traversal. Under XRI resolution, this unioning applies in two contexts:

- During service selection, if a given authority node does not contain an SEP matching the service selection parameters, then Ref(s) are traversed, if any, in an attempt to find an authority node that does match.
- During canonicalID verification, if the given XRD does not contain a canonicalID that is a child of a canonicalID of the authority producing the XRD, then Ref(s) are traversed, if any, in an attempt to find an authority node that does pass the verification test.

How does all this foil the spoof?

Looking back at figure 3, we now know that the solid green edge is shorthand for a red node containing a Ref. When cid="true", the resolver will follow this Ref and return an XRD containing a verified canonicalID only if that canonicalID is along the path of the Ref. One might think of it this way: the Ref actually defines the verifiable polyarchy. In the case of the spoofed (dashed green) edge, we can assume that the authority [@badguy] does not have a red child node containing a valid polyarchical Ref.

Input parameter and red node metadata interplay

At the beginning of this article, I talked about how my EatSourGrass.com web application needs to invoke the XRI resolver on a given authority XRI (such as @xdi*andy.dale):

- To obtain the canonicalID so that it can look up the member account, and
- To obtain the OpenID authentication service SEP so that it can authenticate the member.

To accomplish this, it sets the resolution media type parameter to `_xrd_r="application/xrd+xml"`. This way it can obtain the XRD in order that it can pull out the canonicalID. It sets the service selection parameter `sep="true"` and sets the service selection type parameter to `_xrd_t="http://openid.net/signon/1.0"` so that the resolver performs service selection looking for the OpenID service. Finally, because it wants to make sure that the XRD metadata has not been spoofed by a bad guy, it sets the canonicalID verification flag `cid="true"`.

Table 1 shows the interplay between the Resolver input parameters and the metadata in the XRD returned for the red node shown in figure 5. In this table we use the terms *dereference* and *self reference* as follows:

Dereference: occurs when @ootao*xdi*andy.dale is resolved and the result is an XRD describing the the { 1!3!4 } node in figure 5.

Self reference: occurs when @ootao*xdi*andy.dale is resolved and the result is an XRD describing the red node in figure 5.

For example, this is the XRD returned to the resolver when the resolver invokes the authority resolution service for node [@xdi] with the query *andy.dale.

Red node XRD contains matching SEP?	Red node XRD contains CID?	Resolver behavior
no	yes/no	Dereference: The resolver does not find an SEP match for the OpenID service in the red node, so it follows the Ref and returns the XRD for the Ref target [@ootao*andy] produced by [@ootao].
yes	yes	Self reference: The resolver finds an SEP match for the OpenID service in the red node, so it returns the XRD for the red node [xdi*andy.dale] produced by [@xdi]. Because the resolver is returning the XRD for the node containing the Ref—and not <i>following</i> the Ref—I call this a self reference.
yes	no	Dereference: Even though the resolver finds an SEP match for the OpenID service in the red node, it follows the ref anyway and returns the XRD for the Ref target [@ootao*andy] produced by [@ootao]. This behavior is due to the canonicalID verification rule that says if no verifiable CID is present, Ref(s) will be followed to find them.

Table 1: Resolver behavior for figure 5 with resolution media type parameter `_xrd_r="application/xrd+xml"`; canonicalID verification (`cid="true"`); and service endpoint selection (`sep="true"`).

In other words, under canonicalID verification, if the XRD for the red node contains no SEP matching the OpenID service type, then the red node is dereferenced and the black Ref target node [@ootao*andy] is returned. If the XRD for the red node *does* contain a matching SEP, then the dereference will occur only if the red node does not assert a canonicalID.

Table 2 below illustrates similar behavior when obtaining just the XRD without service selection (sep="false".)

Red node XRD contains CID?	Resolver behavior
yes	Self reference: [# Document this #]
no	Dereference [# Document this #]

Table 2: Resolver behavior for figure 5 with resolution media type parameter `_xrd_r="application/xrd+xml"`; canonicalID verification (`cid="true"`); but no service endpoint selection (`sep="false"`).

Table 1 and table 2 both describe the resolver behavior under canonicalID verification (`cid="true"`). It can be seen how resolving the authority XRI `@xdi*andy.dale` in figure 5 will result in the red node being dereferenced such that the XRD metadata returned is for the node [@ootao*andy] as produced by the node [@andy].

The tables also show that self referencing behavior (obtaining the metadata for the red node as produced by the node [@xdi]) will occur depending upon whether or not the red node asserts its own canonicalID. It is worth pointing out that self referencing behavior can also be obtained by *changing the resolver parameters* to turn off canonicalID verification. For example, if we change table 2 to so that canonicalID verification is turned off (`cid="false"`) then both rows will result in self references, regardless of the whether or not the red node asserts its own canonicalID.

Comparing to XDI

Those familiar with XDI graph modeling may have noticed that figure 4 contains the same shorthand used in XDI to illustrate the polyarchical parent-child relationship using the XDI “Link node” (which also happens to be drawn in red in XDI graphs.) The following lists the similarities and differences between the red node of figure 5 and the “Link node” of XDI.

Similarities between the red node in figure 5 and the Link node in XDI:

- Both type of nodes are used to define polyarchical parent-child relationships. They both define relationships that are traversed during either XRI or XDI subsegment traversal.
- Both type of nodes are addressable. That is, the black edges entering them can have one or more local names (synonyms) that are addressed by XRI authority resolution or XDI addressing. Since both nodes are addressable, the nodes can be either self referenced or dereferenced, depending upon the query.

Differences between the red node in figure 5 and the Link node in XDI:

- The Link node in XDI is formalized as an element in the XDI schema. In XRI the red node is just a node containing one or more Refs. It does not have an explicit element in the XRD schema.
- The Link node in XDI has only a single Ref and no other metadata. The red node in XRI can have any number of Refs and can also contain other metadata such as SEPs and canonicalIDs.

CanonicalIDs and Service Selection

[# **TODO**. Discuss this: This article has discussed using of CanonicalIDs as account identifiers in order support the using of polyarchical relationships yada yada. Explain that a given resource can return different canonicalIDs (from different authorities) based upon how Refs are followed in the face of Service Selection. The EatSourGrass.com application uses canonicalIDs obtained by resolving the authority XRI *for service selection for the OpenID service type*. Other applications may obtain their CanonicalIDs from by resolving for other service types (for example an XDI may obtain a given authority's CanonicalID by resolving for the XDI service type. #]

Summary

My EatSourGrass.com web application is architected to use canonicalIDs as the primary key for its member account allowing it to exploit polyarchical authority XRIs as synonymous i-names without the need to explicitly register each synonym with the application. This article has examined how such polyarchical relationships are defined under XRI resolution, how the resolver can prevent the application from being spoofed, as well as some of the properties of polyarchical authority relationships, such as dereferencing and self referencing.

Incidentally, my EatSourGrass.com website is scheduled for global launch April 1, 2007. So strap on your feedbag and mark you calendar!

; -)

References

[TBD]

