The Unhosted Social Web

Ever since I started working on the unhosted project, people have pointed me to the Federated Social Web movement. This paper is the result of my brainstorm about how we could combine the concepts of the unhosted web on one hand, and the federated social web on the other.

The unhosted web is a candidate successor for the Web 2.0 architecture. On web 2.0, websites have become software products that hold user data safely inside of them, on behalf of the users who contributed this data. The unhosted web architecture does the opposite: it strictly separates application providers from data storage providers: use the application of foo.com, but store your data on bar.com. This obviously requires that foo.com and bar.com agree on some protocols to become interoperable. To this purpose, we published the [Unhosted WebDAV protocol stack](http://unhosted.org/spec/dav/0.1), which defines a simple way for storage nodes to publish a restful nosql storage. Unhosted web apps can then build on this per-user storage to provide functionality, without having to deal with storage requirements.

Unhosted WebDAV

The Unhosted WebDAV protocol stack is based on WebDAV for the storage. A client-side app can easily take advantage of the simple PUT/GET-interface that WebDAV exposes. For the discovery of this storage service, Webfinger is used. So if a user logs into application foo.com with user address mike@bar.com, then the application will look up the webfinger records for mike@bar.com, and find a reference there that indicates where the user can OAuth for the service, and where the actual service lives. The webfinger and webdav URLs need to be served with CORS headers in all http responses, so that browsers know that this cross-origin ajax to the storage node is allowed. On top of all this, the application can optionally implement end-to-end encryption, so that the storage node cannot even see the data it is storing on behalf of the user, and really becomes a commodity. We have created [My Favourite Sandwich](http://www.myfavouritesandwich.org) as a demo app.

OStatus

The federated social web, specifically the OStatus protocol stack, has a slightly different approach to decentralizing the web. It does not address the fact that a user's own data is locked into a specific application provider, but instead decouples the choice of application from a user's choice of friends. This is a great help for the indecisive open source community, which has created not one, but several dozens of social network apps for people to choose from. It is also the

only way to allow people to freely choose the software they prefer. The OStatus protocol hinges on pubsubhubbub for propagation of notifications, discoverable through webfinger, and with several semantic social formats on top of it.

Combining 'Unhosted' and 'Federated Social'

Now let's think about how the two could fit together. Of course, we could add a social network to the federated social web, and implement our specific network with an unhosted architecture. We would actually need some things that are not currently supported by the unhosted webdav protocol, but we'll get to that later. First of all, let's think what this would mean. On the unhosted web, a user uses an app published by foo.com, while all the data involved is stored on bar.com. Bar.com is where the webfinger and identity of the user lives; foo.com only serves static html, javascript and css, which is processed in the user's browser. So all connections to friends on other OStatus-compatible social networks would have to be established from the browser, too.

The trouble is that this is impossible, because OStatus was not designed to be participated in from the client-side. OAuth has been updated in version OAuth2.0 to include a client-side flow, but OStatus doesn't mention such measure. Basically, all Webfinger, atom, and other URLs would need to be served with CORS headers in the http response. It would be exceedingly trivial to add this to the OStatus spec, but it wouldn't be the whole solution.

Leave the milk on the doorstep, please

The second part that is missing is a way to subscribe from somewhere that doesn't have a public interface to put a callback url on. We could implement this using the unhosted storage node as a 'doorstep' hub for the client-side app. This is like when the milkman used to leave the milk on the doorstep, because the doorstep is always available, even if there is nobody inside the house.

The client-side app that runs in the browser would need to have a special relationship with its doorstep hub, in that it can OAuth against it, and tell it to follow certain feeds on its behalf. These feeds would then, instead of pinging the in-browser app (which would be impossible), ping the doorstep hub, which would then aggregate all results, and pass them on to the client-side app either on request, or on long polling. That way the client-side app only has to check one place, and not one for each feed it follows.

Client-side pubsubhubbub

The protocol between the doorstep hub and the client-side app could be arbitrary, but we propose to keep it as a minimal variation on the existing pubsubhubbub protocol. Namely:

- In subscribe and unsubscribe requests:
    - leave out hub.callback
    - set hub.verify to 'token' instead of sync or async
    - set hub.verify_token to the token you received previously through OAuth2.
- Since the browser cannot receive 'publish' pings on a callback url, it should receive it on a long-polling connection, which could be a simple GET to the doorstep hub's URL, probably with 'token' in the query string set to the OAuth token again.
- Even more so than with normal pubsubhubbub setups, publish pings should be queued up at the doorstep hub if the client-side app is not available (when the user is not online).
- It could also be desirable to do a GET to the doorstep hub that returns immediately if no pending 'publish' pings are available. At the time of writing, we are still playing around with this.

We propose to alter the next version of the unhosted protocol stack (which will come out in October 2011) to accomodate a PuSH doorstep hub to which a client-side app can authenticate using OAuth, in the same way as it already authorizes for the webdav data storage. Most other parts of the OStatus stack, like atom feeds and different machine-readable social profile formats are readily implementable on the existion unhosted webdav specification. We would add a few extra entries to the webfinger records.

Salmon

The only other exception is salmon. If my unhosted storage node is going to hold all my data and act as my 'doorstep', then it will have to provide a salmon end point where people can add comments without me being present. So let's add a salmon end point as well. For now, it seems easiest to allow the client-side app to harvest salmon slaps using the same OAuth token again, and leave the specific client-side app to decide which slaps to add into the existing comments feed.

So in version 0.2 of the unhosted protocol stack (we would no longer call it 'Unhosted WebDAV'), the OAuth token would open up access to three URLs:
- /webdav,
- /push,
- /salmon.

We will keep researching how to do this exactly, so that we can have it working before the planned publishing date of version 0.2 of our protocol, which is 5
October.

Commodity hosting

So far, we have just added an extra network to the federated social web. And it is unhosted, meaning that it can benefit from end-to-end encryption to provide true commodity hosting, where you don't have to trust the commodity host that your profile lives on. Well, you trust them to not mess with your webfinger (which cannot be encrypted, but is also not privacy-sensitive). And you trust them not to suddenly delete your account and deny its prior existence. But at least, you don't have to share your private messages with them in unencrypted form.

As far as we know, most distributed social networks that use a commodity hosting architecture and that work from the browser, encrypt private messages only during transport. This is because the javascript that could do the encryption cannot be signed, and could be tampered with by the host that is about to receive the encrypted data. So using JS crypto there would be almost useless. The only way to provide tamper-proof end-to-end encryption without separating the app server from the data server is using browser plugins or java applets. Our unhosted social network would however have one centralized, trusted domain name serving all javascript, and many distributed commodity storage nodes doing 'dumb storage', that's to say, storing encrypted blobs whose content they cannot decypher. But if that were the only advantage, we wouldn't be so excited...

Now for the big trick...

The real excitement (and we thank Laurent Eschenauer for coming up with this concept) is hot-swapping social apps on the same data. Imagine you don't even have to export and import data to move from one social application to another. With the unhosted web architecture, this is the case. So far, this feature hasn't been exploited a lot yet, because the unhosted project is still very, very young, and because it only works if you have a good semantic structure that these hot-swappable apps agree on. Thanks to the federated social web (and also simply thanks to the semantic web as such), social apps have such semantics. If I can read my friend's social data that comes from another app, then I can also read my own social data that was generated while i was temporarily using another app.

This opens up many possibilities for the end-user, and for software development. Imagine some apps are better for photos, and others are better for discussion threads. You could swap from one social app to another, not as a matter of easy migration, but just because you're mixing various social apps. They can no longer even be called different social networks, because they're so interchangeable. So we envisage a move from one social web where each user can choose which app they want to 'live on', to one social web where each user can mix several apps, to each create their own social multi-tool experience.

One feed per identity, or many?

Moving back to techicalities, one question that arises is to what extent you may want to keep different news feeds for different apps. For instance, you may want to try out an app without giving it access to your public photos feed. However, if we define one photo feed per application, then which of these photo feeds should be mentioned in the webfinger file as 'the photo feed of the user'? Can it make sense to allow one user to define more than one timeline, depending on the application (even within the same online identity)? We don't currently have an answer to that question, and welcome discussion on the topic.

Federated-only social surnames

Since unhosted storage nodes only store data, and do not provide functionality, they cannot be called social networks, or even social network nodes. But they do provide identity, and towards the outside (towards other federated social networks), they are indistinguishable from social networks that have a vivid web 2.0 application running inside them. Because the users on a same storage node look from the outside like they are members of a family, even though there is no local 'family life' going on inside (there is no application running which could be called a social network), these federated-only social nodes are more like a sort of outward facing family surname.

As a demo, we started the first federated-only social surname: [Federoni] (http://www.federoni.org). There is no web app running on federoni.org, you need an unhosted social app to use your identity @ federoni.org. We are currently in the process of getting this working - you will know we are making progress if you see the Federoni surname appear on one of the networks that make up the federated social web. The Federoni social surname is reserved for people who contribute (in whatever way) to the unhosted project in general, or the unhosted social web in particular.