Design for a Meta-currency Platform

v0.1rev5
March 26, 2009
Eric Harris-Braun, Art Brock, Adam King, et. al.

Preamble

The meta-currency project is about defining an approach to creating a currency network. Although we describe a particular protocol that implements this approach it should be clear that we believe the network fundamentally not to be an Internet phenomenon, but rather a human one that at this time is using the Internet to implement a larger pattern. Again: although this whole document looks like a design for yet another Internet protocol, it is fundamentally NOT one. Rather it is intended to be an instance that reveals a deeper pattern of human interaction in the context of larger social systems.

Abstract

This document outlines a conceptual metaphor for how to think about designing a meta-currency protocol and some particulars that apply that metaphor to the currency case. The driving concerns that lead up to this design approach are:

1) Simultaneous deployment of multiple currencies of arbitrary design that cover the whole range of wealth acknowledgement (tradable, measurable & acknowledgeable)
2) Separation of concerns: end-user interaction, security, data-integrity, identity, programming languages, state-keeping, etc.
3) Unification through smart-edge, dumb-center network.
4) Full decentralization for robustness, resiliency and maximum sovereignty.
5) Easy integration of legacy systems and approaches
6) Structural support for the "Intrinsic Data Integrity" paradigm.

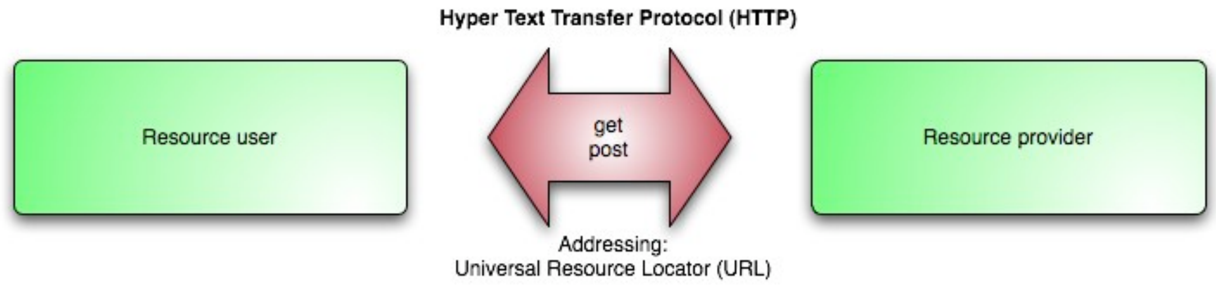
Information resource consumption and provision mediated by a very simple protocol (HTTP) drives a massive increase in the wealth of information resource services. A similarly structured protocol: the Meta-currency protocol (MCP), for mediating between players in an agreement structure (game players) and the current relationships of those players to each-other (game states) will likewise drive a massive increase in wealth of the communities formed by those players. HTTP is the dumb center of a smart-edged network that delivers arbitrary resources between resource provider and consumer. MCP is the dumb center of a smart-edged network that allows game players to make plays in arbitrary games. HTTP uses an addressing scheme of URIs (built on top of domain names and IP addresses) to manifest network endpoints. MCP uses and addressing scheme called the Meta-currency Identifier or MCI (built on top of URIs) to manifest network endpoints. While HTTP can deliver arbitrary resource formats (specified by the Content-Type header of the protocol) practically it major usefullness came because a general purpose data format, HTML, was co-developed with the protocol. Likewise, though MCP is completely game and play format agnostic, co-developed with MCP is a the Simple Game Format Language (SGFL) which provides a jumping off point for immediate development of highly interoperable agreement structures.

A conceptual metaphor

The setup:

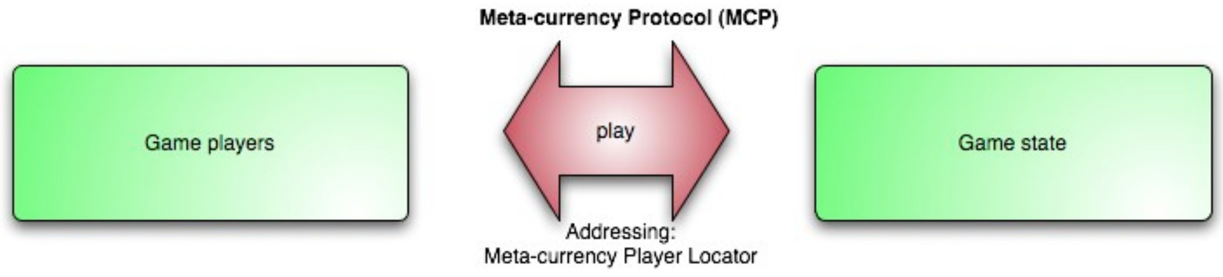
The genius of the Hypertext Transfer Protocol is that it creates an incredibly dumb center for an incredibly smart edged network. It does so, as we've learned from the REST revolution, for

"Resources". The HTTP protocol can be viewed as an intermediary between resource users, and resource providers. On one side of the protocol is the user issuing requests (usually GET or POST) for resources specified by a URI. On the other side is a provider that keeps state relative to the posts, and returns state relative to the get requests, i.e. a Resource. Both sides of the protocol are open both to infinite varieties of expression, as well as evolution. I.e. not only are there billions of web-pages, but also there new types of "web pages" are being developed all the time. On the user side we see the evolution in rendering formats (HTML, XHTML, HTML5, gif, jpg, png, etc.); client side "Turing power" (java-applets, javascript, flash, etc); and devices (web-browsers, cell phones, web-crawlers, etc). On the provider side we see the evolution of generating resources from flat-files to dynamically generated data from RDBMSs, to full Web2.0 semantic mashups. The key to all this fabulous evolutionary power is the unifying power of HTTP which (like the internet itself) creates a dumb-center/smart-edges network. The core pattern to accomplish this is to ignore almost everything about the resources being transferred and instead focus on an addressing space and a small set of actions on those addresses.

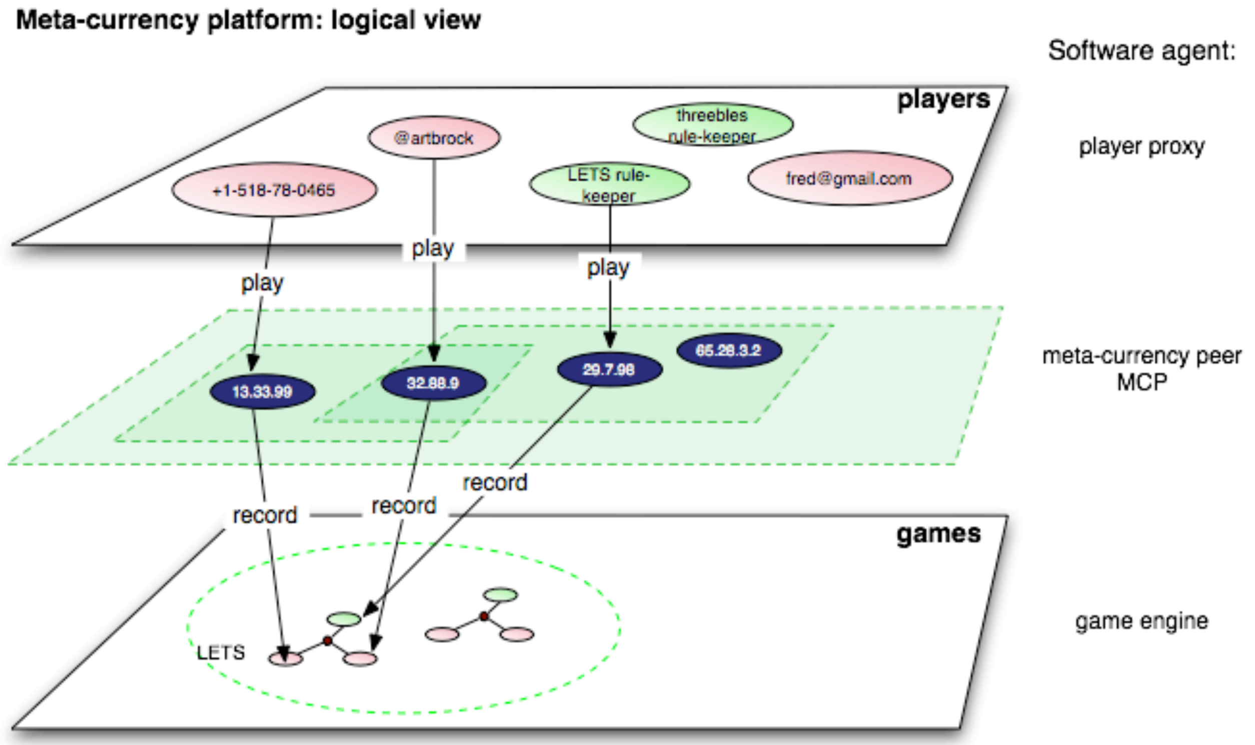

Hyper Text Transfer Protocol (HTTP)
Resource user
get
post
Resource provider
Addressing:
Universal Resource Locator (URL)

Applying the metaphor to currency:

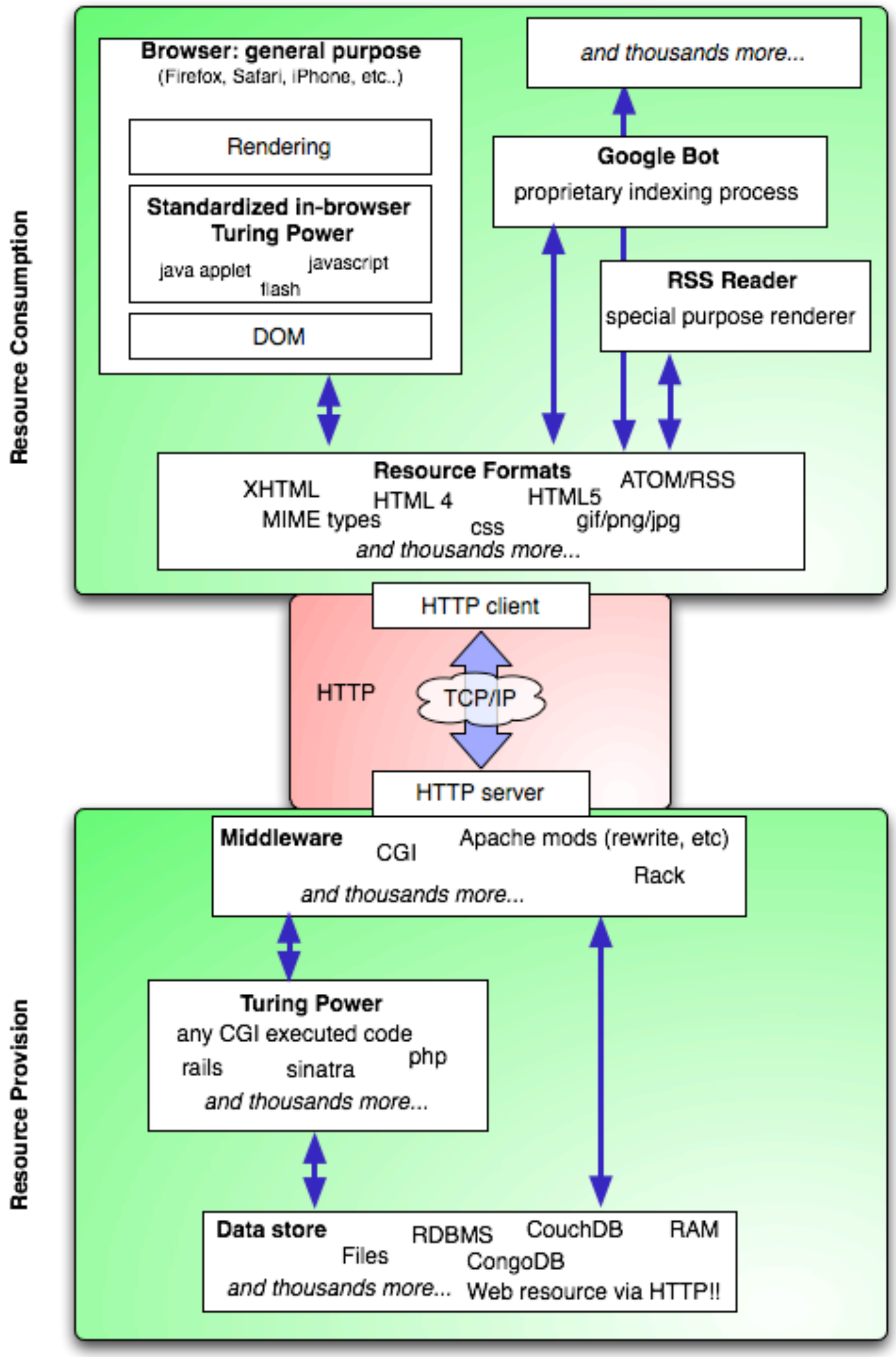
The meta-currency protocol is closely analogous to HTTP, but instead of being about enabling resource availability, it's about enabling "making plays in games". About the game analogy: Currencies can be thought of simply as games that a group of people have decided to play with each other by following a certain set of rules. (The prototypical currency game is one in which people have agreed to track each-other's exchanges of goods and services. Supposedly this should make things equitable because contributions to the community are rewarded with "credit," i.e. money, that can be spent on taking benefits from the community. This should prevent people from taking value from the community without contributing equally. The reality is another matter because some folks get ahold of the credit creation mechanism, but that's a different story...)
Where HTTP links resource "user" and "provider," MCP links the players of the games, and the game boards. Analogous to HTTP which defines a few actions (get, post, put & delete) on an address space of resources (URIs) with arbitrary content, MCP defines a few actions (play & your-turn) on an address space of players making plays of arbitrary contents and type.



So let's break this out into more detail. A logical view of the meta-currency platform reveals players with some identity that can make plays. They do this through peers which are the software agents that execute the MCP. By contrast *web servers* (apache, nginx, etc.) are the software agents that execute the HTTP protocol and bring together resource users (web-client) and the resource producers (web application frameworks/services, etc). *Meta-currency peers* are the software agents that execute the MCP protocol and bring together game players and game state. Game players and the game state also have software agents that manage them, *player proxys* and *game engines*. Here's a logical view of this architecture:



Lets compare this with the HTTP and web. Here is a more detailed diagram of how HTTP brings together resource provision and consumption:



There are two central aspects of this diagram that are taken and apply to the Meta-currency project:

1- and thousands more...
This is almost the whole point of HTTP. It allows the development of those "thousands more" by the smart edges of the network. That's precisely the goal of MCP in regards to the patterns of agreement that people play by.

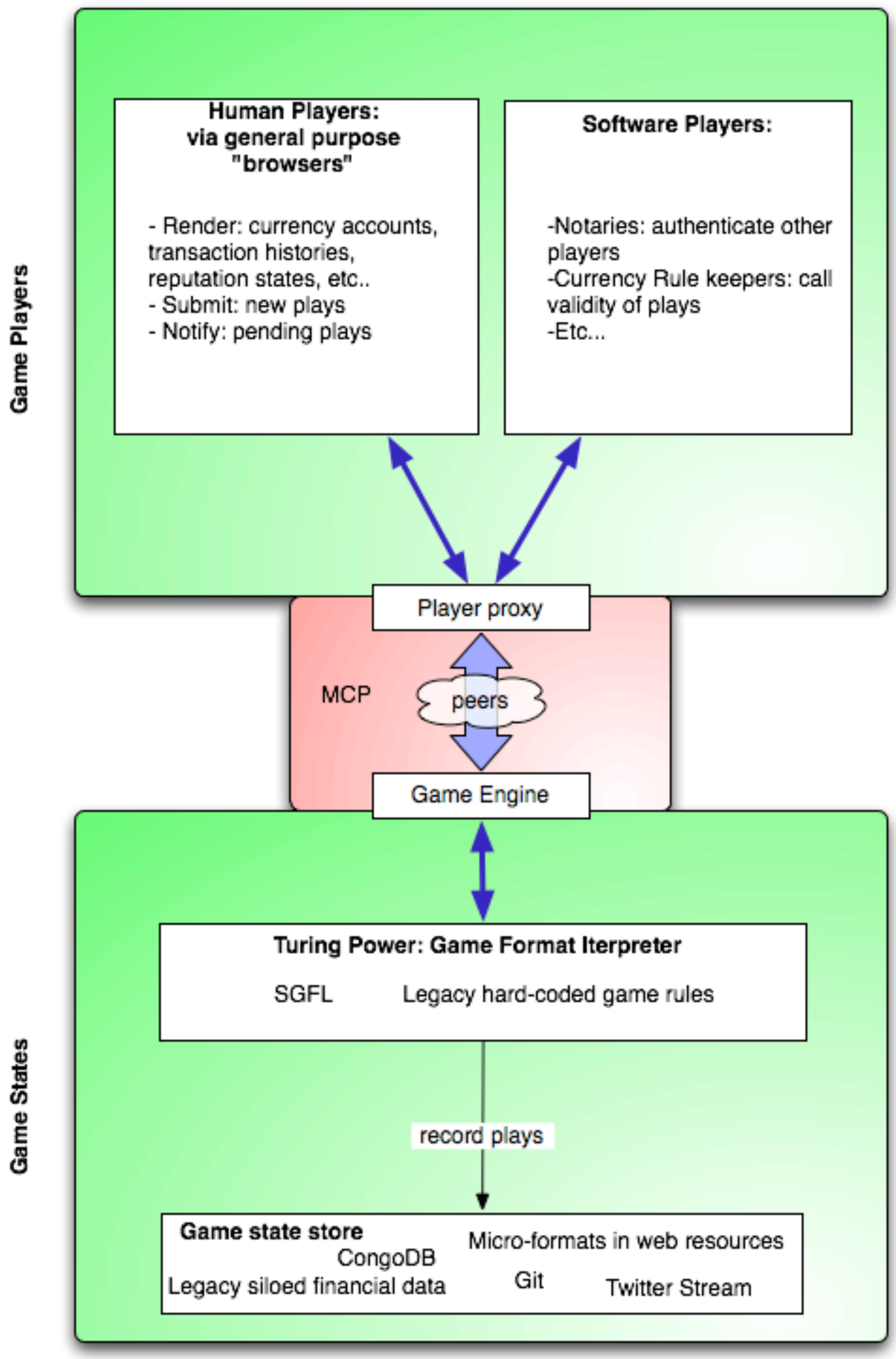
2- Formats: MIME & HTML -- XML Schema & SGFL
Resources types transferred over HTTP are specified with MIME in the Content-Type header. This simple feature allows HTTP to carry arbitrary resource types. MIME was an existing resource type specification language which was simply used by HTTP, however co-incident with the development of HTTP was a new language for representing the resources that was designed to play well in the network context that HTTP created. That format is HTML, the prime feature of which was a format for referencing other resources via the <a href> and <img src> tags.

MCP follows a similar pattern. It is designed to allow plays in arbitrary games to be submitted by players anywhere on the network. The structure of a play so submitted is defined by a document in any XML scheme language (DTD/RELAX NG/W3C XML Schema). The particular schema for the play structure is thus equivalent to the MIME type of a resource for HTTP. Thus MCP can act as the conduit between players who want to play games, and the engines that hold game states. But, just as HTTP was co-developed with HTML as a base use-case; co-developed with MCP is a "game format" for representing possible games which, like HTML, is similarly designed to play well in the network context. We call this format: SGFL (simple game format language). SGFL is an XML language which has provisions to reference other players and plays in the game space. The contents of the games specified in SGFL can be Turing powered by arbitrary programing languages, similar to how resources formatted in HTML are Turing powered by languages like javascript with the <script> tag, as well as flash objects, or java applets. Just as HTML evolved we expect SGFL to evolve.

This structure of using 1) any XML Schema to specify play format, and 2) a particular XML language as starting language for game specification; allows MCP to meet the design goal that it be able to integrate well into legacy systems. For example, a legacy currency system hosted in a silo behind a meta-currency peer could easily accept plays in OFX, and at the same time a more generalized game engine that can interpret an SGFL specified game will be able to realize the full distributed potential of the meta-currency network.

See appendix 1 for some initial ideas on how SGFL will look, including example games definitions.

With these two core aspects in mind, we can draw a similar diagram to for MCP and the game player/state equation:


Game Players
Human Players: via general purpose "browsers"
- Render: currency accounts, transaction histories, reputation states, etc..
- Submit: new plays
- Notify: pending plays
Software Players:
-Notaries: authenticate other players
-Currency Rule keepers: call validity of plays
-Etc...
Player proxy
MCP
peers
Game Engine
Game States
Turing Power: Game Format Iterpreter
SGFL
Legacy hard-coded game rules
record plays
Game state store
CongoDB
Legacy siloed financial data
Micro-formats in web resources
Git
Twitter Stream

MCP in action

Play Examples

MCP is not a "transfer" protocol for games specified in some format like SGFL. It's not about sending games specified in SGFL around a network (that will happen with regular old HTTP). Rather it's about using games that have been defined in SGFL (or other formats) to interpret plays, i.e. update game state. So lets look at MCP in action using a play specified by SGFL. A play in the basic_LETS currency specified by the SGFL in appendix 1 might be submitted to a peer like this:

```
<mcp-play>
 <players>
```

```
    <player mci="234.2938.383">
     <credentials>
     <credential type="OAuth">...</credential>
    </credentials>
   </player>
   <player mci="324.1643.3233" />
   <player mci="2342.33958.23" />
 </players>
 <action type="play">
 <game schema="http://lets.org/basic_lets">22398.3834.2122</game>
 <content schema="http://lets.org/basic_lets/plays/payment.dtd">
  <play>
    <from>234.2938.383</from>
    <to>324.1643.3233</to>
    <amount>1000</amount>
    <aggregator>2342.33958.23</aggregator>
    <memo>test transaction</memo>
  </play>
 </content>
</mcp-play>
```

The components of any MCP play are Players, Action type, Game and Play content. The first three of these components are like the headers in HTTP, and the last is like the body.

The content of a play is completely arbitrary. The peer that that handles player specified by the play is expected to hand the content to the game engine, and it is expected to know how to interpret the contents of the play in regards to the given game specified. Thus the content of a play could just as well be in any other format, i.e. for a legacy system a play might be specified in OFX.

Creating new games and new players is out-of-band to SGFL and MCP. However it may be the case that the object-model of for SGFL in a particular game engine will allow creation of new games or players, just as posting a resource through HTTP may create a new URI accessible via HTTP.

Adding players to games is in-band in MCP however. This is similar to calling "new" on an object type in an object oriented language. To do this we specify the add-player action type rather than the play action type.

```
<mcp-play>
 <players>
    <player mci="33429.22.138">
    <credentials type="Basic">
     <password>X1934sdkasie2</password >
    </credentials>
   </player>
 </players>
 <action type="add-player">
 <game schema="http://IssuedCurrency.org/ourcurrency">33429.22.138</game>
 <content schema="http://IssuedCurrency.org/ourcurrency/plays/_new.dtd">
  <play>
    <class>member</class>
    <starting_balance>100</starting_balance>
  </play>
 </content>
</mcp-play>
```

MCP Play Dynamics

[In this section I will describe the details of how a play is injected into the network via a peer, how the peers communicate with eachother, and hand off plays completely signed plays to game engines to transform the states]

When this play is injected into the meta-currency network by being submitted to a peer (kind of like when an e-mail is given to an SMTP server for delivery), what the peer has to do is quite straight forward:

1) resolve the addresses of the players in the play.
2) deliver the play to the peers currently holding state for those addresses.
3) for peers holding state: evaluate the credentials presented for the players, and, if appropriate, pass the play to the game engine which then evaluate the state transforms for the play, and apply them.

Intrinsic Data Integrity

The meta-currency platform is designed to support from the outset the paradigm of Intrinsic Data Integrity, which moves from us from the concept of security by data protection via putting up strong walls, to security by data incorruptibility via the "physics" of the system. This is a crucial approach when the intent is not to rely on silo-ed data stores and to allow for peered/distributed data.

Background

There were times when whole villages hid behind castle walls and great moats, and when the borders of nations were defined by defensible boundaries like rivers, coasts, mountain ranges and Great Walls. Closing something off behind high walls and tight defenses is certainly one approach to security.

However, we've discovered that there are much more effective methods to ensure security (even if many of us are not conscious of it). Notice that today, there are very few people living behind castle walls, and few nations focused on defensible borders. We’ve replaced walls with laws and moats with tort. Rather than erecting walls to hide from hordes of savages, it turns out we’re much safer having people internalize rules and laws. They suddenly cease being “savages.”

But when it comes to dealing with computers and data, we’ve mostly ignored this lesson. We lock our data behind passwords, permissions and firewalls, relying on that one type of security when we could have the data internalize rules which keep it secure. There are a few instances where we do this like when we use CRCs (Cyclic Redundancy Checks) to ensure a downloaded file hasn’t been tampered with, parity bits in RAID arrays, or check sums on network packets. However, we don’t generally structure the data we use for decision-making ways that ensure its integrity.

One great example of software that does do this is the distributed source code management tool called git. Instead of a tightly controlled central repository where you protect from unauthorized people committing changes to it, git allows anybody to download the source code for a project (for example, the kernel of the Linux operating system), make modifications to the code and then commit the code again!

This would be very dangerous to do with typical software or databases. Imagine if you could download your bank’s internal software or your bank account data, make changes to it, and then recommit it again. Everybody would be changing their bank balances all the time. With git you can’t alter something without adding a completely new commit which records your signature. If you tried to make a sneaky unsigned change to existing data, it would longer match with the signatures attached to it. This makes it safe for people to change to the Linux kernel and yet ensures they cannot introduce flaws or security holes because it will break the signed data patterns.

By implementing Intrinsic Data Integrity, you can allow people to have a completely authoritative copy of their own transaction history. This data could be specifically shared with the parties of each transaction, or shared with third party notaries or auditors, or possibly open to public review. New ways of analyzing and aggregating the data become possible, because the original data is available in a linked and signed chain. Just like you can notate the moves of a chess game and replay them to recreate the state of a chess game, you can replay the signed transactions to create the state of an account or the whole currency system at any point in time.

Intrinsic Data Integrity Support in MCP

Intrinsic Data Integrity support is built into MCP in that it is expected that all plays will carry signatures from each player in the play, that they agree to it. This approach is simlar to how we close deals in a contract negotiation. What makes a valid contract is all the proper signatures on the contract, including perhaps a notary, or even including filing it with a government office, where the signature isn't a human one, but an institutional one in the form of the stamp. This makes contracts verifyable. This approach is supported in MCP and SGFL, where players add their signatures to a play, and when all signatures are in place (as specified by the game rules) then the play is considered complete.

User Interfaces

In the meta-currency project we hope to learn some lessons from the past regarding the issue of separating semantics from content presentation. We all suffered through a long period where HTML was not a semantic language, but was used for presentation and rendering of the content of a resource was done directly in HTML with tables and all kinds of tricks with spacer gifs and so on. Fortunately most rendering tasks for the purposes of human consumption have been moved to CSS, a classic case of separation of concerns. The meta-currency network will require lots of human interface, for example rendering data entry fields for a play on a hand-held device, or listing a transaction history on a web-page. So likewise just as the actual mechanics of the state transformation is handed off to the game engine, so we hand off rendering to mechanisms such as XSLT which can transform the XML specified in a play into necessary HTML for rendering in a web-browser. In the case of using XSLT, it can be either specified in the SGFL itself, or referred to indirectly by it, just as HTML can embed CSS or point to a css style-sheet at a different URI.

MCP Addresses & Namespaces

Background

Central to any network is addressing and namespaces. Addressing creates a space for entities to be located. All networks have an addressing scheme; the postal network, the phone network, the internet, etc. Without an address, there is no network. There are many approaches to addressing in the currency network domain. Paypal took on the idea of using email addresses. Credit card networks issue carefully designed numbers where the first digit identifies the card type (Mastercard, VISA, AMEX etc). Banks have routing numbers for that allow checks to make their way through the Federal Reserve clearing system. Modern networks often have several layers of addressing schemes that help in the goal of separation of concerns. IP addresses are addresses for routing packets to machines. Domain names are addresses for people to own and thus to associate identity to machine readable resources. i.e. the mapping of domain names onto IP addresses serves to separate the two concerns: packet routing and ownership/identity. Another separation of concerns between the two levels of domain names and IP addresses, is that of "human-friendly" and "machine-friendly." Humans find it hard to remember IP numbers strings, but easy to remember names that map well into our language space. A third benefit of the two level system where domain names map to addresses is that the "logical location" of a resource is assigned by domain name, which allows the "physical location" to be changed over time. This kind of indirection provides a crucial level of practical resiliency and utility.

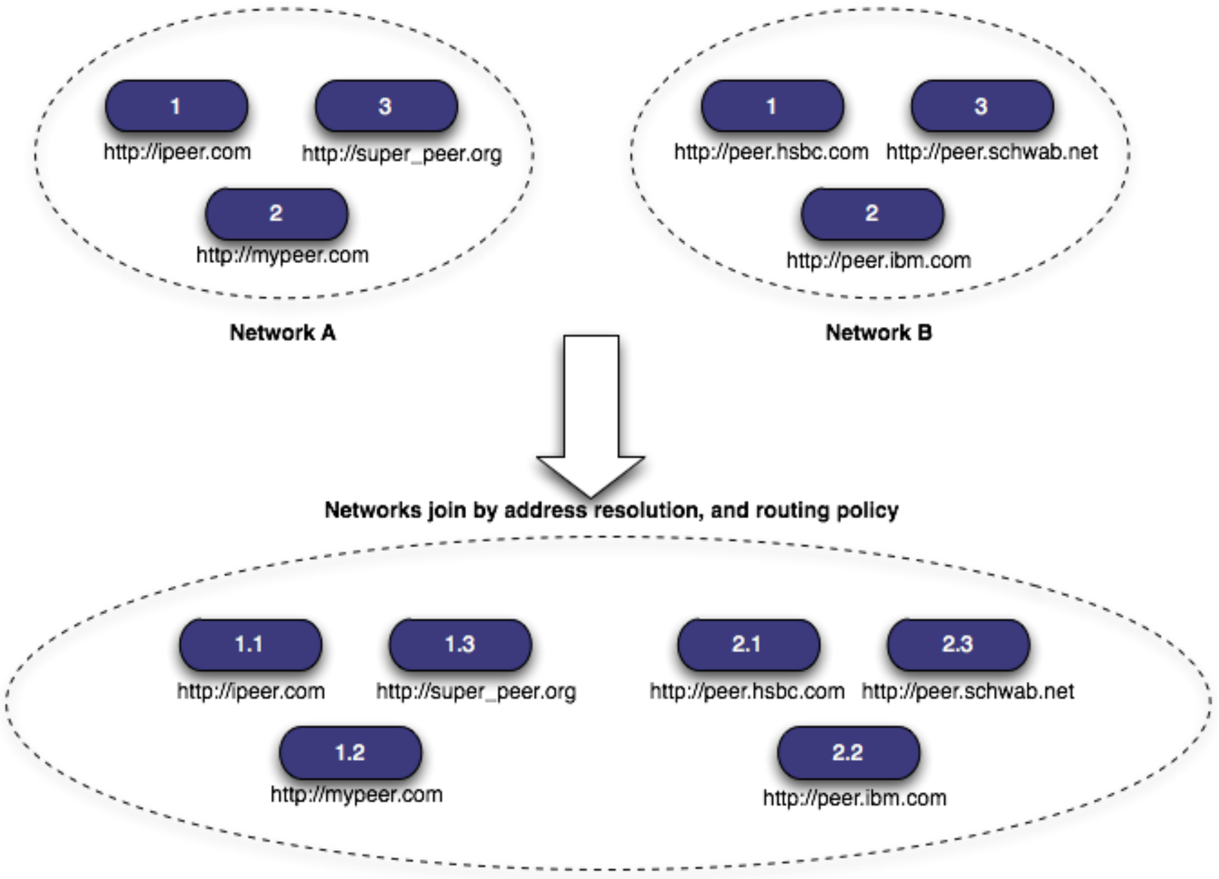
MCP approach and assumptions

The meta-currency project's approach to addressing: 1) Identity: MCP assumes that determination of the "true" identity of players will be an ever evolving process, so it does not build in identity into its addressing scheme. 2) Location: MCP does not assume that players are directly accessible on the Internet. They may be behind telephones, carrier pigeons, etc, thus an MCP address is always considered a proxy for some other service that handles interaction with the "true" player, thus there is a layer of MCP addresses that are abstract and mapped onto internet addresses for use on the Internet. 3) Sovereignty: the MCP addressing scheme is designed around "inverted hierarchies", which allow for mutual sovereignty between individuals and groups.

We also take a two level approach to addressing: 1) Meta-currency identifier (MCI). This address locates a player on the network abstractly. MCIs are the equivalent of domain names for the Meta-currency network. 2) URI. All MCI resolve to a URI. The URI is a pointer to the peer(s) handling that MCI at given time. These URIs are equivalent of IP addresses for the meta-currency network. Just as an IP address of a given domain can change, so can the URI of a given MCI change. But MCI are unique.

Inverted Hierarchies

Internet domain names are classic top-down hierarchies. ICANN manages the top of this hierarchy, and all names live under it starting with the top level domains. This model replicates the classical notions of ownership of a territory. The meta-currency project implements a looser, more flexible strategy for generating address space hierarchies. The idea is simply that: 1) there is no top level & 2) any group can associate with another group to form a new level above the two of them. Thus the hierarchy emerges from cooperative behavior rather than being enforced from the top. The spaces that emerge then tend to be more about joint stewardship of a common resource, than ownership of a territory.



[More about MCI resolution software agents and specs for the different layers.

Conclusion

We have described an approach to designing a meta-currency platform that we believe has sufficient expressive capacity for communities to design all currency systems that address all levels of wealth generation. The approach meets our driving concerns.

Appendix 1-- SGFL

SGFL allows for the specification of the following fundamental entities: game, player class, state, play, state transform. The game element is the root element which specifies a game that can optionally inherit from some other game. Player class, specifies all the different types of players in a game. State specifies the data structures that will be maintained on a game-board for each player type. Play specifies the structure of all the possible plays in a game, including data formats, and it also specifies the transformation on player state that the play will cause.

Below is an example game specification in SGFL for a LETS currency. A very simple LETS currency really only needs one player class with one state i.e. to represent the members of the LETS and their balances. This LETS currency is slightly more complicated because it keeps track of player transaction volume and also includes another player class, an "aggregator" that has a state to hold aggregate volume of a group of members. This currency has two possible plays, a "payment" and a "reversal." (Note that it is not possible to use a payment of a negative amount to implement a reversal because the volume is always calculated on the absolute value of the amount)

```
<game name="basic_LETS">
 <player_classes>
  <player_class name="member" />
```

```
   <player_class name="aggregator" />
  </player_classes>
  <states>
   <state name="member_state" player_class="member"><integer name="balance" /><integer
name="volume" /></state>
   <state name="aggregate_volume" player_class="aggregator"><integer name="volume"
/></state>
  </states>

  <plays>
   <play name="_new">
    <fields>
     <field type="" id="class">
    </fields>
    <script type="ruby">
     @member_state.balance = 0
    </script>
   </play>
   <play name="payment" player_classes="member">
    <fields>
      <field type="player" id="from" />
      <field type="player" id="to" />
      <field type="player" id="aggregator" />
      <field type="integer" id="amount" />
      <field type="string" id="memo" />
    </fields>
    <script type="ruby">
     @from.member_state.balance -= @amount
     @from.member_state.volume += abs(@amount)
     @to.member_state.balance += @amount
     @to.member_state.volume += abs(@amount)
     @aggregator.aggregate_volume.volume += abs(@amount)
    </script>
   </play>
   <play name="reversal" player_classes="member">
    <fields>
      <field type="play" id="play_to_reverse" />
      <field type="string" id="memo" />
    </fields>
    <script type="ruby">
     @play_to_reverse.from.member_state.balance += @play.amount
     @play_to_reverse.from.member_state.volume -= abs(@play.amount)
     @play_to_reverse.to.member_state.balance -= @play.amount
     @play_to_reverse.to.member_state.volume -= abs(@amount)
     @play_to_reverse.aggregator.aggregate_volume.volume -= abs(@amount)
    </script>
   </play>
  </plays>
</game>
```

You will note that the SGFL above implies that game engines will need to have something similar to how browsers implement the Domain Object Model for HTML. Any instance of a running script will have available to it representations of the fields specified in a play, and the states associated with a player. In the example of the reversal play above you see how the script can access the contents of a "play_to_reverse" to be able to excecute its transforms.

SGFL and legacy currencies:

For the purposes of the meta-currency protocol, the minimum necessary entity to be specified is simply the plays. Here's what an SGFL game for a legacy payment system that uses OFX to specify all plays, might look like:

```
<game name="legacy_payment_currency">
 <player_classes>
  <player_class name="payer" />
 </player_classes>

 <plays>
  <play name="payment" player_classes="payer" type="external" />
   <format>ofx</format>
  </play>
 </plays>
</game>
```