The Live Web Series

From Personal Computers to Personal Clouds

The Advent of the Cloud OS

Phillip J. Windley, Ph.D., *Kynetx*

Other Authors in this Series
Craig Burton, *KuppingerCole*
Scott David, *K&L Gates, LLP*
Drummond Reed, *Respect Network Corp.*
Doc Searls, *The Searls Group*

April 2012

The Live Web Series *sets forth a vision for the future of the Internet and our interactions on it. This paper is the first paper in that series. Future papers in this series will build upon the ideas presented here to show how personal clouds form a foundation for richer and more satisfying online applications and experiences.*

There's no doubt that clouds are big business. A 2011 report from research firm Gartner (http://www.gartner.com/it/page.jsp?id=1739214) put worldwide software as a service (SaaS) revenue at $12.1 billion, a 20.7 percent increase from 2010 revenue of $10 billion. The SaaS-based delivery will experience healthy growth through 2015, when worldwide revenue is projected to reach $21.3 billion. SaaS is just one component of the overall cloud services sector which a 2010 Gartner Research report projected to grow to $148.8 billion by 2014.

While much of the growth in cloud computing is in the enterprise space—especially infrastructure and platform plays like Amazon and Rackspace—there is significant activity in the area of personal cloud computing. Point solutions like Dropbox along with more holistic offerings like Apple's iCloud and Google's suite of products are all part of the personal cloud space.

Through their personal cloud offerings, companies like Apple™, Google™, and Facebook™ vie for the attention of Internet users. Apple is slightly different than the other two since their users *are their customers*. Apple uses their cloud offering as a way to make your Apple devices more useful. On the other hand, Facebook and Google are literally after your attention because user attention is what ad-supported companies sell to their customers, the ad buyers.

While offerings in the personal cloud space provide real utility for users, they are limited in several important ways.

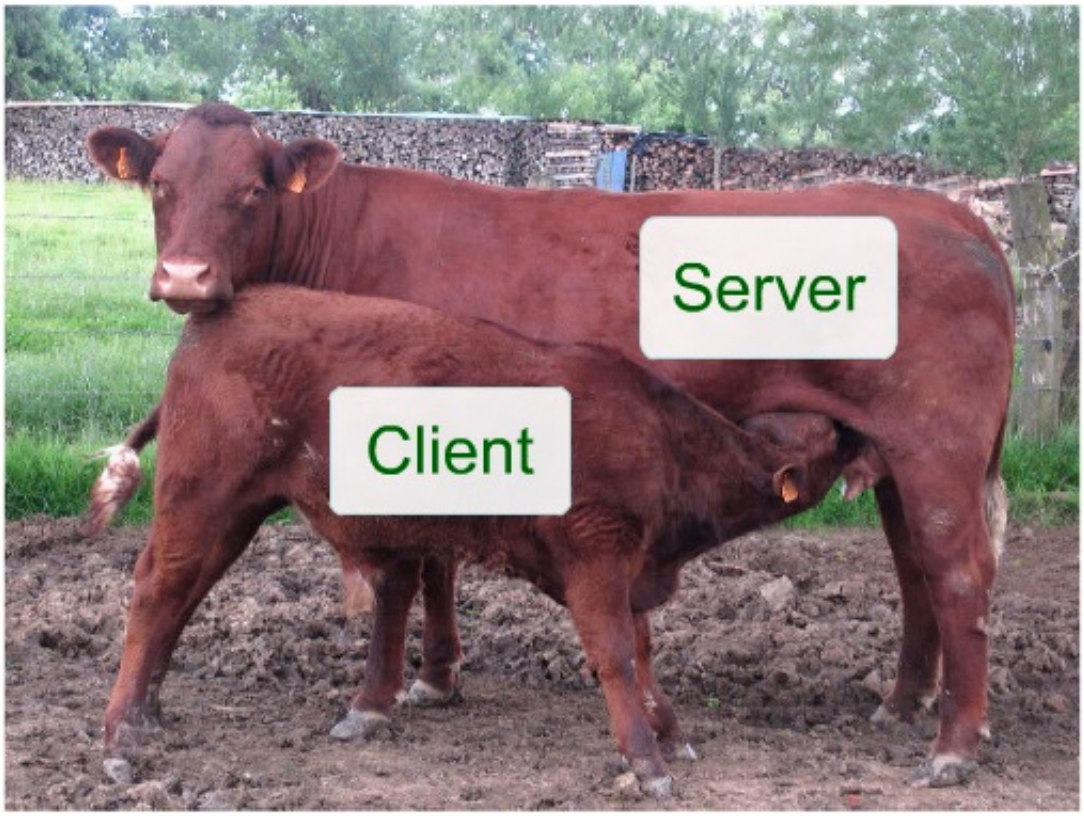
First, *each of these offerings is an appliance*. In other words, they provide specific functionality with only limited features. Think, for example, of the difference between a DVD player and a tablet computer. The DVD player is certainly a computer, but it's a computer with a specific function—an appliance. In contrast the tablet is nearly infinitely extensible and will take on different capabilities as its owner updates it, installs apps, configures it, and links it with other services.

Second, *these personal cloud offerings are silos*, standing alone and apart with only limited interconnectivity. Users might be able to export documents or import calendars, but users do not have unfettered access and use of their data. Interactivity is implemented on an "as the market demands" basis rather than according to the user's individual needs.

We contrast the current view of personal clouds with one where personal clouds are *the successor to the personal computer*. In the personal-cloud-as-personal-computer model, owners of a cloud control it in the same way they control their computer. They decide what apps to install, what services to engage, and how and where the data is stored.

For a better understanding of why this is desirable and leads to increased opportunity and power, we need to explore how we interact online today and how we could do it differently. In original conception, the Internet was a network of peers where each spoke TCP/IP and had an IP address. (Aside: We're stretching things slightly here and ignoring the idea that the Internet was a network of networks with TCP being only one of many transport protocols. But that was more of a political sop that helped gain acceptance of IP than anything real.) Of course, now most things "on the net" have non-routable NAT'd addresses. The world would be much different if the original architecture had persisted.

The result of this shift is a split between "servers" where the *real work* is done and "clients" that are dependent on servers. Doc Searls has labeled this the cow-calf model (http://blogs.law.harvard.edu/doc/2011/04/02/a-sense-of-bewronging/) .



The Calf-Cow Model

Doc and David Wienberger (http://www.hyperorg.com/blogger/) wrote about this eloquently in 2003 in an essay entitled World of Ends (http://www.worldofends.com/) . In that essay, they say:

> When Craig Burton describes the Net's stupid architecture as a hollow sphere comprised entirely of ends (http://www.searls.com/burton_interview.html) , he's painting a picture that gets at what's most remarkable about the Internet's architecture: Take the value out of the center and you enable an insane flowering of value among the

> connected end points. Because, of course, when every end is connected, each to each and each to all, the ends aren't endpoints at all.
>
> And what do we ends do? *Anything* that can be done by *anyone* who wants to move bits around.

Email is a good example of this principle at work. Email is delivered on the 'Net using the SMTP protocol. Phil has an email server that runs in the cloud that understands SMTP. It functions as his "end" for email online. What's better, he didn't have to stand up a Linux box and install Sendmail to get it. Google runs it for him and he just uses it: email-as-a-cloud-service.

Personal Clouds are Virtual Machines

Personal clouds that are akin to personal computers will give people other "ends" on the Internet. This more expansive vision of what a personal cloud can be is empowering because personal clouds will

- change how we relate to everything in our lives
- rearrange how we buy and sell products and services
- revolutionize how we communicate with each other

You might think that this is a tall order. But we think there are good arguments why many of the futures we're expecting are best supported—maybe can *only be supported*—by personally controlled, general purpose computers that operate in the clouds. Let's take them in turn.

Change how we relate to everything in our lives—As more and more products and services go online, our capacity to control them and interact with them will depend on the compute power we can bring to bear on the problem. The current model is to turn the device into a server and build an iPhone app as it's client. For example, is this is how the Nest thermostat works. That works great when there's only a few or until you want two of these products to start interacting with each other. The future will require *any-to-any* interactions that can't be supported by special purpose apps on a smartphone. Applications running in the cloud, however, can use protocols to mediate the interaction in a more general way (http://www.windley.com/archives/2012/03/ways_not_places.shtml) .

Rearrange how we buy and sell products and services—For new conceptions of online commerce like VRM (http://cyber.law.harvard.edu/projectvrm/Main_Page) to work, people need independent, autonomous agents who will represent them in transactions. The only viable place we can imagine these agents operating is in some kind of personal cloud.

Revolutionize how we communicate with each other—Giving everyone customizable, computationally complete ends on the Internet also provides us with more flexible means of interacting with each other and the services we use. A personal cloud can have an infinite number of incoming and outgoing "channels" for communication, meaning that you can dedicate one or more channels to anyone or anything. Independent channels have three important properties:

- **Management independence**—If a contact starts spamming you with messages you don't like, you can simply delete the channel and they're gone—without impacting any other communication channel you've established.
- **Permission independence**—You can give different channels to each contact and set priorities and permissions on the allowed notifications.
- **Response independence**—If two vendors have different channels to contact you, you can treat them differently and even contextually. For example, you might want to see commercial messages from flower shops only in the weeks leading up to Valentines Day, Mother's Day and your wife's birthday and then only until you purchase something.

Personal clouds needn't be complex to set up or manage for them to have significantly more power than the cloud appliances that companies offer people today. We envision systems that are no more complicated than a smartphone that offer the features and benefits described above. Indeed, such systems exist today.

The Personal Space

Lately there've been a lot of terms thrown around related to personal data: *personal data store, personal data service, personal data locker,* and so on. All of the forgoing terms, however, miss the key concept of *doing.* The companies and projects creating technology in this space are all after the same goals: giving people more control of the their data and empowering them to do new things with it.

We distinguish between the ability to securely store and use personal data from the ability *do something with data*. After all, if you're going to create a place to store personal data, most of the value lies in making use of it. KuppingerCole calls these systems *Live Management Platforms*. KuppingerCole focuses on the idea of *informed pull* as a complement to *controlled push* as a distinguishing feature between a Life Management Platform and personal data stores.

Our use of *personal cloud* is meant to convey a similar distinction between a place where data is security stored and a place that acts for the user. Certainly, personal data will be a key component of a personal cloud—managing a modern life requires enormous quantities of data. Users might choose various and multiple places to keep their personal data. A personal cloud should be able to use, manage, and protect all of them. But a personal cloud is more than a personal data store in *exactly the same way* that a personal computer is more than a file system. A personal cloud is a virtual machine that operates 24/7 on behalf of it's owner. We anticipate people having multiple personal clouds. The remainder of this paper will describe the architecture that we envision for these personal clouds, using the metaphor of a PC operating system as a road map for understanding the development of an operating system for the personal cloud.

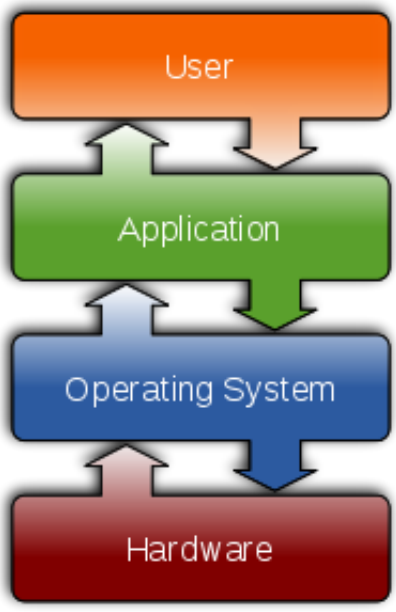
Personal Clouds Need an Operating System

If you need an OS for you laptop, your phone, and your tablet, why don't you need one in the cloud? Our current conception of how people use the cloud has significant limitations that could be mitigated with the introduction of a cloud-based operating system that people can look at as their "virtual computer" that's always on, always working.

The IBM PC with PC DOS

Long ago there was something called a “microcomputer.” The name seemed natural enough given that they were smaller than the “minicomputers” of the time. The first examples were crude machines that required significant technical expertise of the user. As microcomputers became easier to use, they took on a new name based on the feeling that they produced—they became “personal.” One of the hallmarks of a “personal computer” is its flexibility. Personal computers are capable of doing what their owner wants and doing different things at different times.

This flexibility is provided by a utilitarian piece of software call the “operating system” that sits between the computer and the software it runs. We’re all familiar with operating systems because they give the personal computer it’s personality and define most of what the user thinks of as “the computer.” Operating systems run programs, manage stored data, and provide common services like printing or access to networks.



Operating systems create a virtual machine for the applications

Today's cloud offerings aren't nearly as utilitarian as a personal computer. A personal computer—be it a desktop, laptop, tablet, or even smartphone—is vastly more powerful and flexible than that Apple's iCloud, Google Docs, or Facebook—despite the latter's programming platform and API. This isn't simply a matter of waiting for Google Docs and other personal cloud offerings to mature. Their architecture has a fundamental limitation by not having an operating system. Let's explore why.

Operating Systems

We wouldn't be stretching belief to call the operating system the most important piece of software on your computer. The operating system sits between the hardware and the applications. The operating system presents a virtual machine that is orders of magnitude easier to program and interact with than the bare metal of the hardware. For example, here are a few things that operating systems do:

- **Identity**—All modern operating system provide identity as a fundamental service of the operating system, tracking users, permissions, preferences, and other important attributes. Our conceptions of how computers work are based on this knowledge of our identity.
- **Program execution**—operating systems present a method of running programs that is very different from the underlying machine. First, the OS keeps track of metadata showing that the data *is* a program. Lower down, the OS is responsible for loading, threading, and other details of how programs actually get run. At the UX level, the OS is presenting a view of applications that enables user selection, configuration, and control.
- **Data abstraction**—operating systems present a view of data that is very different from how it's actually stored. For the most part, disks have tracks and sectors. Getting data on or off the disk involves knowing specific locations of the data and stringing it together into the entire object. What's more, the disk doesn't store any metadata as a matter of course. The disk doesn't know if the data it's returning is a JPEG or an MP3 file. Finally, the disk doesn't have any concept of data ownership and permissions. An operating system, in contrast, provides applications with an abstraction that let's data be accessed in a hierarchical manner and provides all kinds of metadata so that applications know what they're getting, when it was created, who owns it, who can read it, who can write it, and so on. The operating system manages permissions and access to data.
- **Communications**—operating systems similarly abstract communications. Sockets, among the most primitive of OS communication services, are orders of magnitude easier to use than working with the Wi-Fi or networking hardware directly. And of course, operating systems present even more abstract ways of communicating when you consider the myriad network services present in any modern OS. What's more, not only does the OS present a high-level view for programmers, but also presents the user with interfaces for easily managing various parameters associated with networks.

Without an operating system, each application has to perform all of these complex tasks themselves making the application more difficult and costly to create. Moreover, as a consequence, each applications will end up performing similar functions differently, making them difficult to use. Imagine, for example, having to configure the network for each application using different concepts, interfaces, and so on. The operating system provides a virtual machine that presents a consistent interface to both users and programs and is both easier to use and easier to develop applications on.

Cloud Operating System

None of today's popular personal cloud offerings are as powerful or flexible as a personal computer, but they could be—*if they had an operating system*. A cloud operating system (COS) would be capable of the following:

- keep track of identity information, attributes, and preferences for the owner
- run unlimited applications of the owner's choice

- store and manage the owner's personal data no matter where it resides
- provide generalized services that any application can take advantage of
- mediate and abstract the usage of Web-based APIs—the libraries of the cloud

A cloud OS provides a individual, independent compute space in the cloud for everyone. This is in stark contrast to the Web 2.0 model where individual applications are hosted on various servers with the browser or mobile phone app as the integration point. The Web 2.0 model leaves much to be desired:

- individual applications are silos with little ability to interact with each other
- relying on apps on mobile devices or browsers on computers leads to differences in configuration and user experience and creates an over-reliance on synchronization
- services—like contacting the user—are reimplemented in each application resulting in a fractured user experience and endless configuration woes

In contrast, a cloud OS solves these problems by creating a space where computation happens in the cloud, data access is abstracted to break down silos, and common services are always available. In the Cloud OS model, mobile devices and computers look like very powerful peripherals that mediate user interaction.

Let's explore how a cloud OS can provide a place for user-controlled applications to run in the cloud, abstract data access, and provide common services for any application to use.

The Foundational Role of Identity in a Personal Cloud

If we're to build personal clouds supported by a cloud operating system (COS), then we need to understand the key services that the COS would provide to the user. Operating systems are not monolithic pieces of software, but rather interlocking collections of services. One of the most important things to figure out is how a cloud OS can mediate an integrated experience with respect to authorized access to distributed online resources.

The concept of identity is foundational in modern operating systems. In Linux, for example, user IDs (`uid`) and group IDs (`gid`) are used by the kernel to determine file and device access as well as process ownership and control. User names and passwords are just the means of reliably setting the user ID so that the kernel can determine access levels.

From this simple, foundational identity system and its associated access control mechanisms grows a whole virtual world that you think of as "your computer." When you log into a machine, you are presented with an environment that makes sense of your files and your programs. You control them. They're arranged how you like them. The system runs programs just for you. You're work environment is highly personalized.

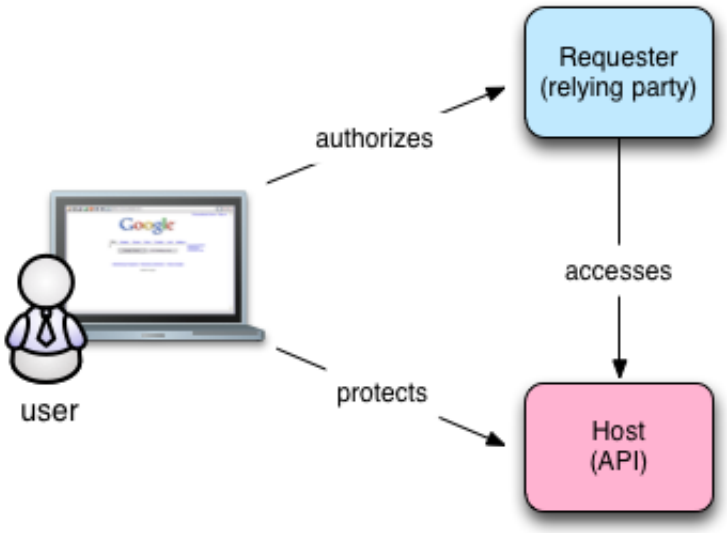
We don't see this same kind of integrated, virtual experience when we're online because there's no foundational concept of identity online. We struggle to provide context between different Web sites. Numerous standards from OpenID to OAuth (http://oauth.net/) have been developed to bridge these gaps, and we've come a long way, but they are not enough—not yet. In short, people have lots of identifiers on multiple Web sites, but they have no overarching identity context, no presence. Bridging these various systems to provide integrated access control has been one of the most important areas of technology development over the last five years.

One of the interesting developments in access control for distributed online resources is a project called User Managed Access (http://kantarainitiative.org/confluence/display/uma/Home) or UMA. For purposes of this blog post we're going to simplify it—and OAuth—a great deal, so if your an

identity expert, bear with us. UMA extends the ideas behind OAuth in several key ways. The one we want to focus on is its introduction of a critical component that provides the overarching authorization context we need in a cloud OS.

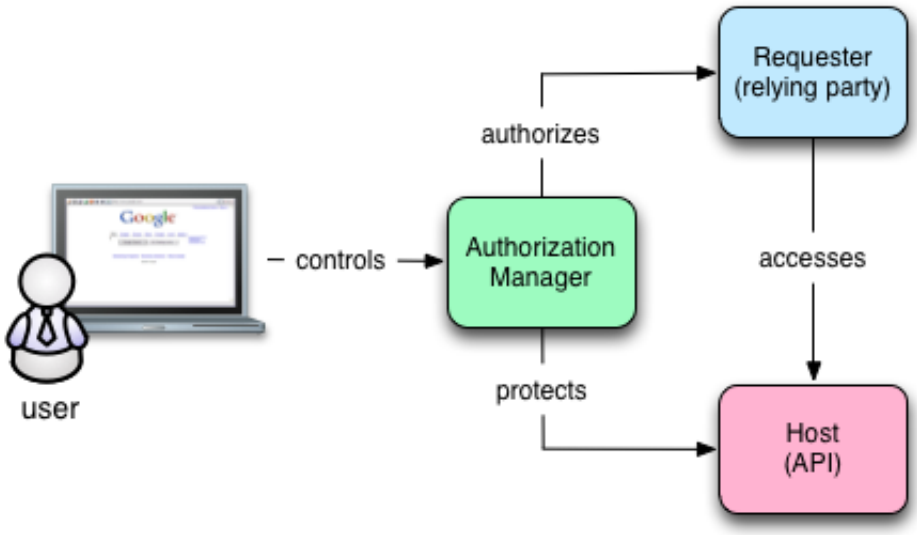
We would misconstrue UMA and OAuth if we were to compare them too closely to the identity and access control systems in Linux. The online world is vastly more complicated because it is distributed rather than centralized with myriad identity systems and methods for controlling access. Recently OAuth has had great success in creating a standard way to control access to APIs. We credit OAuth with much of the success APIs have had over the last several years. Still OAuth suffers from some limitations that we think will keep it from playing the role we want in a cloud OS.

To see why, let's look more closely at OAuth. You've undoubtedly used OAuth to link services on one Web site with another. That's its primary use case: Site A, the "requester" wants access to a resource being hosted on Site B—usually behind an API. The requester needs your authorization to access the resource. You get redirected to the host where you're asked if you approve. If you grant access, a relationship is hardwired between the host and the requester. You can revoke the access at any time at the host. The picture looks something like this:



OAuth users interact with the requester and host directly

In contrast, UMA provides another critical piece that acts on behalf of the user called the **authorization manager**. Consider the following picture:



UMA provides an authorization manager to act for the user

The *only difference* in these two diagrams is the authorization manager. In the UMA diagram, the user controls the authorization manager and it interacts with the requester and the host. The authorization manager represents the user in the authorization transaction and its subsequent management. Think about all the places you've used OAuth. Can you remember them all? Would you know you had to sever a connection between Facebook and Twitter if you wanted to stop some obscure status update behavior? Maybe not.

In the OAuth diagram, no system represents the user's interests. Instead, the user is responsible for bridging the context between the sites as well as remembering what's been authorized and where. With UMA's authorization manager, the user has one place to manage these kinds of interactions.

An authorization manager of some sort will be a key component in a COS. The authorization manager is *active, rather than passive*. The authorization manager contains policies that allow it to act as the user's agent, even when the user isn't present. Rather than a single, hardwired connection, the authorization manager can be continually consulted and access can be granted or denied based on changing conditions and contexts.

Don't make the mistake of thinking this is just about identity. In fact, it's not really about identifiers and associated attributes at all. Identity is important because it forms the skeleton of data. The issue is how identity allows permissioned, controlled access to distributed resources. We don't have to unify identifier systems—although we may need to abstract them—to achieve our purpose. One of the key features of UMA is it's ability to offer fine-grained access control to any online resource—not just APIs—regardless of the underlying identity systems and their credentials. For a complete discussion of how UMA, OAuth and OpenID Connect are related and where they differ, we recommend this blog post from Eve Maler, the force behind UMA (http://blogs.forrester.com/eve_maler/12-03-12-a_new_venn_of_access_control_for_the_api_economy) .

Regardless, when we contemplate an UMA-mediated experience from the user's perspective, we think they'll view UMA as providing a personal context to their online interactions. They'll view that personal context growing out of themselves because something knitted their various, fractured online identities together. The same real-world person will be at the heart of all this regardless of the various fractured credentialing and permissioning systems that underlie it. Just as the personal computer operating system helped create a coherent computing experience in the pre-networked environment, so too can similar arthitectures provide that greater cohesion in the networked environment. That greater cohesion is pereived by the user as greater "identity integrity", and indeed it provides a more reliable interface for users with other resources and

users in the network.

UMA and its provision for an authorization manager is exactly the kind of development that highlights what kinds of utility a cloud OS provides. UMA is just one example of where people are starting to realize that users need systems that act on their behalf to help them manage their interactions. People need systems in the cloud that represent them and what they want done. Having Web sites and APIs is not sufficient (http://www.windley.com/archives/2012/03/ways_not_places.shtml) . A cloud OS goes beyond mere hosting to a system of autonomous agents acting against policy to represent people in the cloud.

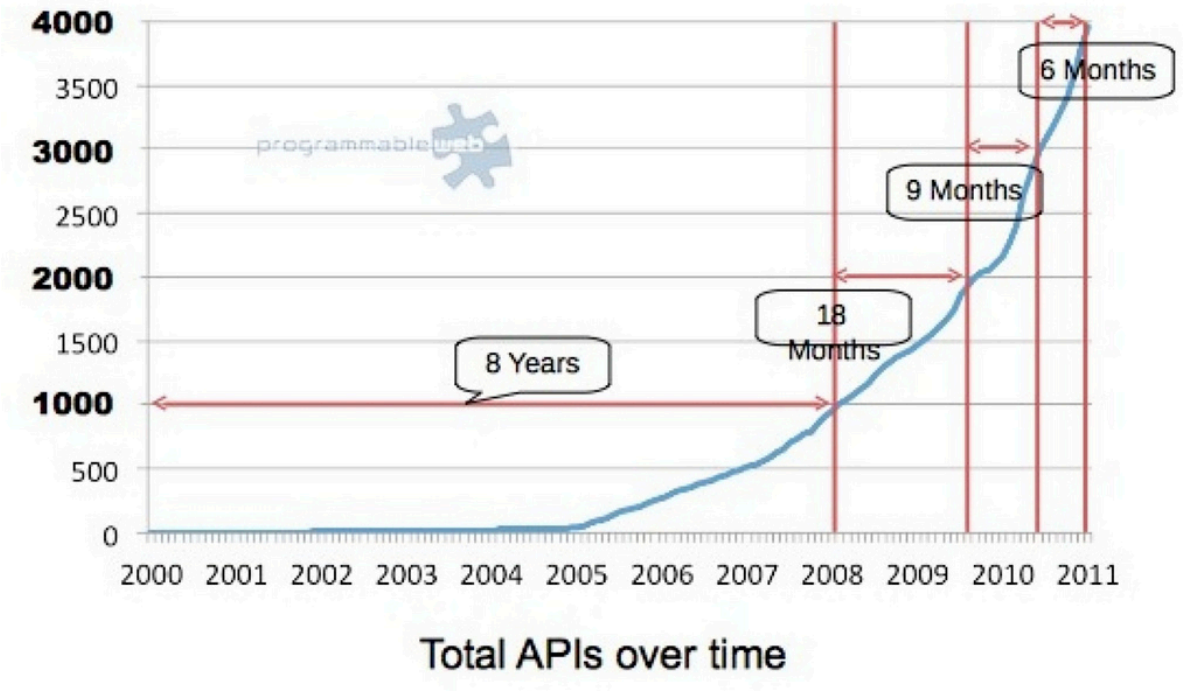
Of course tackling authorized access to online resources is only the first step. Our interest is in using these systems to control online data and programs so that the user effectively has a virtual online presence that represents her, manages her data regardless of where it may be stored and provides real leverage, so she can get more done with less effort, risk, and cost.

Data Abstractions for Richer Cloud Experiences

As we discussed earlier, one of the primary services of a cloud OS (COS) would be data abstraction. Traditional operating systems provide data abstraction services by presenting programs and users with a file system view of the data stored in the sectors of the disk.

We have the same kind of data abstraction opportunities in the cloud, although we aren't talking about translating sectors into files, of course. And like the access control problem we discussed earlier, the problems that a cloud OS faces are made more complex by the distributed location and control of the data we want to access.

As the following chart from Programmable Web (http://www.programmableweb.com) shows, the growth of the number of APIs has been exponential over the last 12 years.



The growth of APIs has been exponential over the last 12 years

All these APIs present a tremendous opportunity for application developers and they've taken advantage of it. Many of the most interesting applications we've seen in mobile and online over the last few years involved mash-ups between multiple APIs.

But with that abundance of data comes a problem: programmers have to learn the various details of the APIs, their access methods, and error codes. If you're just concerned about a few that you need to create a particular app, that's no problem and thus the API economy has flourished. But there are legitimate uses of all this data require using not only multiple APIs, but also APIs you may not be aware of as you're writing your application.

Let's take a simple example: a phone number. Suppose your phone number is stored at Facebook, LinkedIn™, Google, and several other places around the Web. A developer writing an application to run in your personal cloud needs access to your phone number. What should she do? Right now, there are the following options:

- The developer can store your phone number in the app, giving you yet another place where your phone number is stored. The downside is that when your number changes, there's one more place you've must remember to update. If you forget to change your number in the app, it stops working.
- The developer could pick an API, say Google, and just tell people they *have to use* Google. This works as long as everyone is comfortable using Google.
- The developer could choose to support a number of APIs where phone numbers are stored and give users a choice. The downside is that the more APIs the app supports the harder maintenance becomes.

The problem is exacerbated as the number of data elements increases—especially as they need to come from different APIs. We talked about the issues around authorization that this causes in the preceding section. But the problems don't stop there. There are two important issues beyond authorization that we need to address if we're to abstract data access and make the developer's job easier:

1. How do we know *where to get the data*?
2. What is the *format of the data* and *what do the elements mean*?

Solving the first requires **location-independent references**—when a program needs access to the user's phone number, location-independent references provide an abstract means of finding where that data is stored. So, for example, suppose you store your phone number in GMail contacts and your friend stores theirs at Personal.com. The application doesn't have to know that or how to connect to those various services. The program references a name that means "user's phone number" and the data abstraction layer in the COS takes care of the messy details.

The solution to the second involves **semantic data interchange**—suppose the program wants the user's phone number but one API stores it as "cell" and another as "mobile." How do we know that's the same thing? For one or two things, it's easy enough to create *ad hoc* mappings; but that quickly gets old. The data abstraction layer makes these translations automatically. Moreover, there can be multiple formats that are used for storing phone numbers.

A functional COS should provide the means (i.e. protocols) for performing location independent data references as well as semantic data interchange. This abstraction layer can ensure that the authorization, location, and semantic issues are dealt with in a consistent way that is easy for the developer and the user. There has been much work on this problem over the last decade ever since Tim Berners-Lee, James Hendler and Ora Lassila proposed the Semantic Web in the May 2001 issue of Scientific American (http://www.scientificamerican.com/article.cfm?id=the-semantic-web) . While we acknowledge that much of what has been done in the name of the Semantic Web has seemed overly complicated to developers of modern Web services, we believe that we're beginning to face the exact problems that the ideas behind the Semantic Web were

designed to solve.

Our choice for a protocol to provide semantic services to the COS is XDI. XDI (XRI Data Interchange) is a generalized, extensible service for sharing, linking, and synchronizing structured data over the Internet and other data networks using XRI-addressable RDF graphs. XDI is under development by the OASIS XDI Technical Committee (http://www.oasis-open.org/committees/xdi) . XDI was created to solve the aforementioned problems in a way that is:

- **understandable**—XDI does not require pre-defined data schemas for new types of data to be exchanged
- **contextual**—The concept of context is built directly into the XDI graph model, so identity, relationships, and permissions can be context-dependent
- **trustable**—XDI identification, authorization, and relationship management are integral features of the graph model and protocol
- **portable**—An XDI account can be moved to a different host or service provider without breaking links or compromising security or privacy

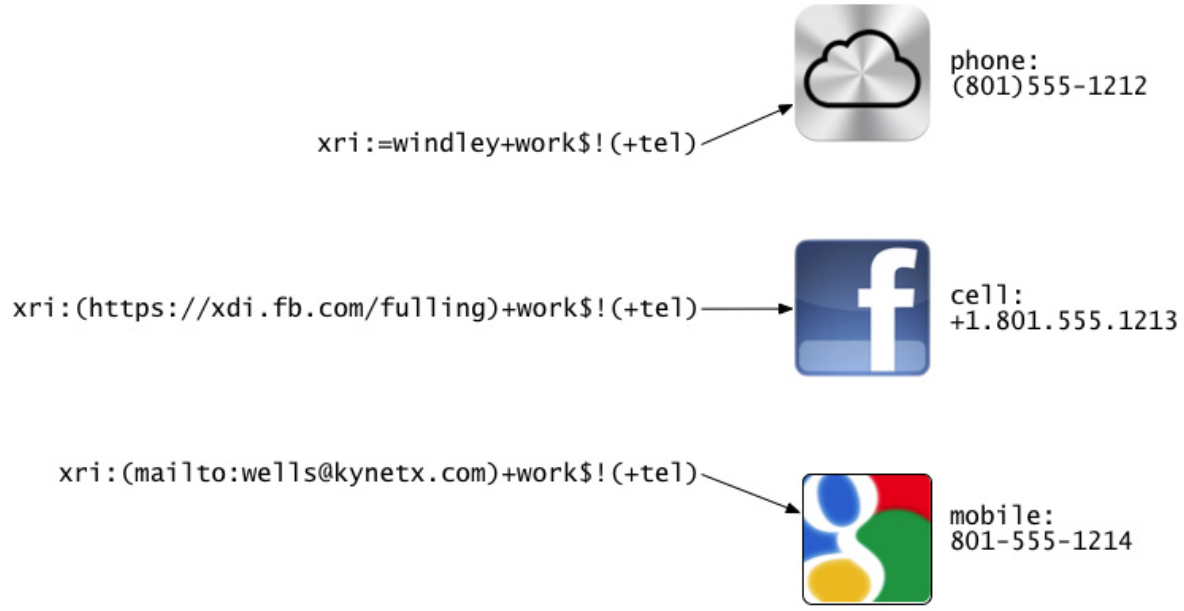
To see how XDI can help, let's continue the phone number example. A developer using XDI to reference the user's work phone number in a KRL program might write something like this:

```
user = get_user_iname();
user_work_phone = xri:#{user}+work$!(+tel)
```

Note: The `#{user}` syntax shown above is meant to convey the construction of an XRI statement using previously calculated data with a KRL beesting. Some other means of constructing XRIs might ultimately be selected as we make concrete progress on integrating XDI in KRL. If `get_user_iname()` returned `=windley`, the resulting XRI reference would be `xri:=windley+work$!(+tel)`. The `+work` clause provides a context for the phone number. The `$!(+tel)` clause specifies that we want a single instance of a phone number, not a multi-valued collection (in the case there's more than one).

Location-independence is the easier property to discuss, so let's start there. Resolving a reference like `xri:=windley+work$!(+tel)` isn't much difference in theory from how a domain name like `www.windley.com` gets resolved. There is a set of known top-level authorities who know how to determine who or what `=windley` is. From there, you (literally) follow the graph to the node represented by `xri:=windley+work$!(+tel)`. That node could reference a data value in any API.

Of course, this kind of independence doesn't happen for free. There's no magic way to know that you keep your phone number at Facebook and your friend has theirs on iCloud. But, a COS could, based on standard mappings, know how to access a user's profiles on various services and provide the right link regardless of who's running the program once the user has given her COS access to her data at the services she uses.



Dereferencing XRIs leads to different locations

Note: The first example in the preceding figure uses an i-name that has been registered with an XDI registry, similar to the way you would register a domain name today. But XDI does not require the use of XDI registries. You can address any data that's available at any URI that hosts an XDI endpoint. This could be any webserver, as shown in the hypothetical example of Facebook supporting an XDI interface, or it could be an XDI endpoint discoverable through an email address using OpenID Connect. All of them work equally well, because once the discovery process reaches an XDI endpoint, all of the data behind it is addressable using XDI.

Who's providing all these XDI endpoints? Ideally the API owners, but that doesn't have to be the case. Think of the XDI endpoints playing the role that drivers play in a traditional OS. If you add a new kind of disk with a different interface to a computer then you need the corresponding driver.

The mapping process shows the power of semantic data interchange. Once maps between common concepts like the user's phone number and it's location in various APIs are made, they can be reused over and over again. If the API changes, changing the map in one place updates it for every application and every user.

Moreover, maps can link common semantic concepts so that we know that *cell* and *mobile* are the same. Semantic mapping solves three important problems:

- Poorly defined semantics—an example might be incomplete phone numbers that assume a context, like a country code.
- Same syntax, different semantics—we might run into data elements that are formatted like phone numbers, but aren't.
- Different syntax, same semantics—this occurs frequently since different APIs use different string formats for the same concept, like phone numbers.

The good news is that we don't have to boil the ocean to get started. Semantics has been made way too mystical and unapproachable. This is really nothing more than the kinds of techniques a good programmer would use to solve these problems, but standardized. A COS could provide mappings for common data elements and common APIs, like contact data or calendars, and make those available to developers. Adding just a few of the most common required elements to the COS would greatly simplify many applications that need access to personal information.

COS-level data abstraction makes programs easier to write and use because:

- Developers don't have to understand the intricacies of multiple APIs.
- The COS manages authorization issues freeing developers from managing the code and allowing users greater visibility into and control over how data is used.
- The COS provides a consistent configuration experience for users.
- Developers don't have to write code to manage configuration.

For data to be useful, however, it must be manipulated. We must be able to write programs that run in personal clouds.

A Programming Model for Personal Clouds

As we discussed earlier, when personal clouds begin to act as peers with other network services, people gain unprecedented power and leverage. Personal clouds can change how we related to everything in our lives, rearrange how we buy and sell products and services, and revolutionize how we communicate with each other.

For these changes to take place, personal clouds must be able to do more than store personal data and mediate interactions with—as important as that is. Your personal cloud must run applications for you, under your direction.

When we say "run programs" you might be tempted to think of the kinds of applications that you run on your laptop or tablet: word processors, spreadsheets, and games. But it turns out there's already plenty of places to do that in the cloud and having those things run in your personal cloud isn't going to change much in your life.

Instead, we're talking about programs that interact on your behalf with other online systems. Those programs end up looking more like services in an OS. For example, you might have an application that manages incoming notifications (http://www.windley.com/archives/2011/12/notifications_in_a_personal_event_networks.shtml) and forwards them to you in a channel you prefer based on context and content. Or you might have apps that intermediates transactions with online merchants. You'll have dozens, event hundreds of apps that help you manage and control your personal data.

We believe that the most natural programming model for a cloud OS (COS) is event-driven. Events indicate something happened—a state changed somewhere. Events often mean that an application should act. Events provide a powerful way to create a metaprotocol that can be used to define the various interaction scenarios (http://www.windley.com/archives/2012/03/protocols_and_metaprotocols_what_is_a_personal_event_network.shtml) that will be present in a personal cloud.

Events augment the traditional request-response programming model of the Web with one that drives action independent of the user. This is a critical component of any system that promises to upset the power structure of the client-server model. David Siegel eloquently explains why in *Apple and the Cloud: A Cautionary Tale* (http://www.xconomy.com/san-francisco/2012/04/06/apple-cloud/?single_page=true) :

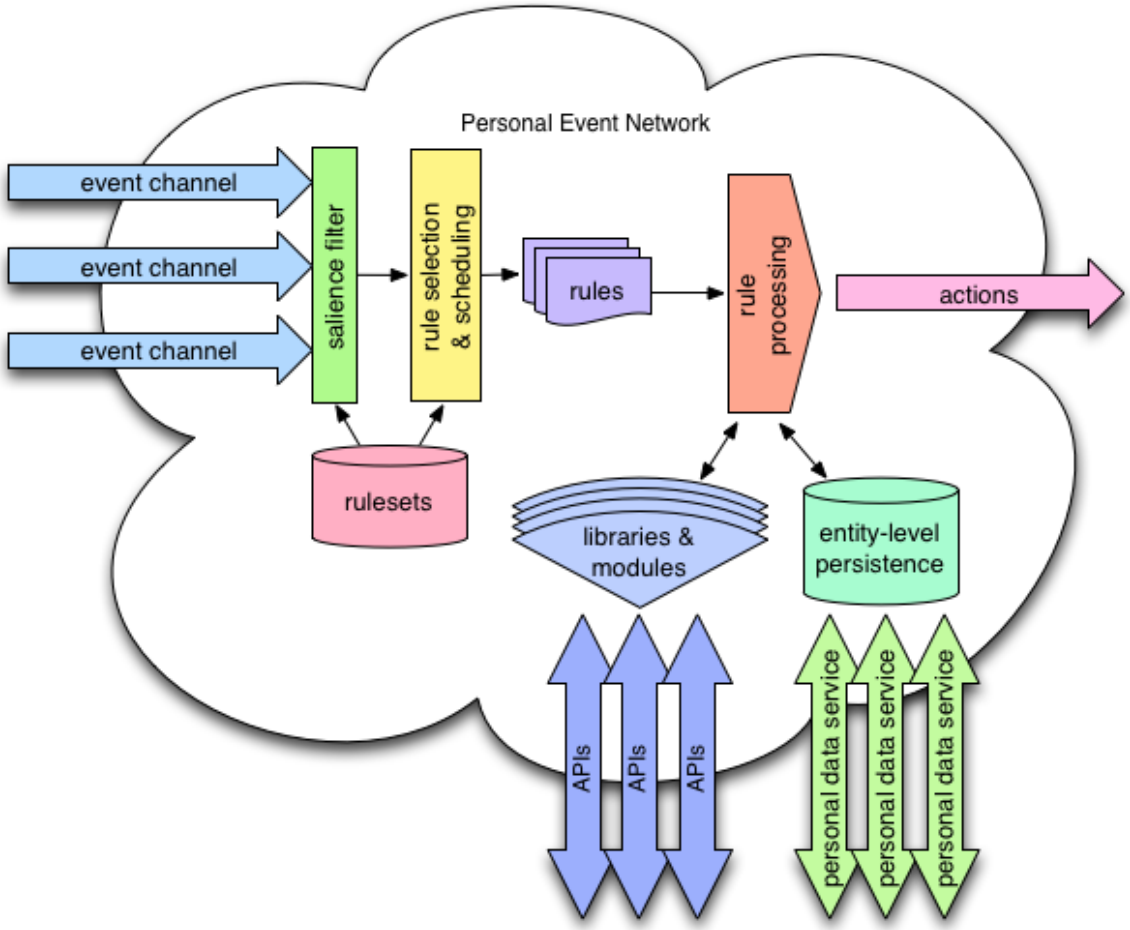
> Adaptability is event driven. It's very different from the demand-driven systems we have today. If something happens in front of you, whether you're on a bike path, driving down the freeway, or flying at 30,000 feet, the system (all participants and their equipment) adjusts. When you take a pill, don't take a pill, hit a golf ball, reschedule an appointment, get in your car, or walk near a store that has something on your shopping list, the event triggers a response and keeps other people up to date automatically. In an event-driven world, we don't know which apps we need, and it won't matter. A piece of code sitting in the cloud that is perhaps almost never used is nevertheless ready to respond to something

unusual, and we may only learn about this software service after we needed it. An event-driven world is designed to change as the data changes.

As David points out, event-driven architectures are adaptable in ways that demand-driven architectures aren't. Making events work in the cloud on behalf of people requires and event-based programming model specially designed for that task.

Personal Event Networks

The event-based programming model for the COS is called a *personal event network* (PEN). *Personal* because each person—entity, really—has their own. A *network* because there is an interconnected collection of programs interacting in the PEN via event-based protocols.



Personal Event Network block diagram

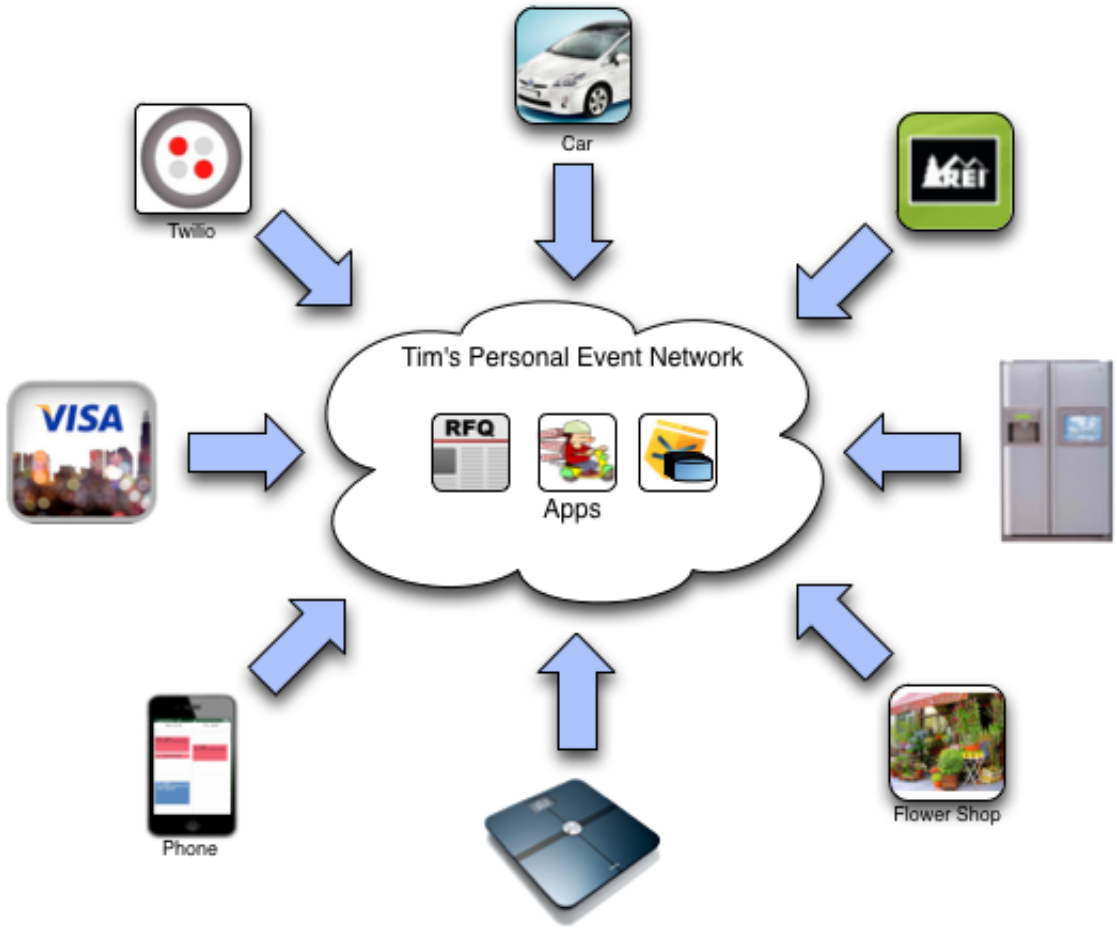
The basic unit of execution in the PEN is a rule. Rules connect events to actions. Applications in the PEN are collections of rules, or rulesets.

The primary duty of the personal event network is scheduling rule execution based on incoming events. The PEN does this by calculating an *event salience tree* for the PEN based on the rulesets that the owner has activated. The event salience tree allows the PEN to quickly determine which rules to execute for any given incoming event.

The PEN listens for events on *event channels*. A given PEN can have a virtually unlimited number of event channels, allowing them to be assigned individually to event generators. This gives the owner of the PEN complete control of inbound events. If an event generator isn't

behaving as the owner would like, the event channel can be shut down without affecting any of the other relationships that the PEN has.

The following diagram shows events coming into a PEN on various event channels:



Event channels raise events into the PEN

You can see that events might be coming from many different devices and services. Only a few apps are shown. The Twilio service might be raising events around incoming phone calls or SMS messages. REI, Visa, and the flowershop might be raising events about purchases or outstanding orders, you car might be raising events about needed maintenance or its location, and so on.

While the preceding diagram only shows three, there might be hundreds of rulesets installed in the PEN watching for various events. Multiple rules can respond to the same event and rulesets needn't necessarily coordinate their responses to a given event. The architecture lends itself to loosely couple collections of rules.

KRL: Linking Events and Actions

The PEN sets up a programming model based on events. Certainly we could respond to events in any number of ways, but we have created a programming language, KRL, or the *Kinetic Rule Language* (http://developer.kynetx.com/display/docs/Manual) , specifically for building applications in the PEN.

KRL has several features that are carefully designed to make writing applications in the PEN easier:

Rule-based programming model—Rules are a natural way to respond to events. Rules in KRL are structured using the *event-condition-action* pattern. That is, rules link events to actions. *When* a particular event occurs, *if* certain conditions are true, then take an *action*. The event and action are fairly obvious components, but don't overlook the condition. Conditions allow rules to take into account context, rather than simply responding to what happened. Rules represent composable chunks of functionality.

Event expressions—Rules are not limited to responding to only simple events. KRL contains an event expression (http://developer.kynetx.com/display/docs/Event+Expressions) language that allows developers to specify complex event scenarios. For example consider the following KRL event expression:

```
select when web pageview
                  "/support/(\d+)" setting(issue\_number)
     before email received
                    subj.match(re/issue\_number/)
```

This event expression indicates that the rule should be selected when a particular Web page is viewed before an email is received. The expression can be thought of as filtering two event streams, those for Web pages being viewed and those for emails being received.

But there's more to this expression than a simple temporal relation between two event streams. Notice that the `issue_number` is being set for the `web:pageview` event based on the page's URL and tested when the email event is received. This event expression is doing the equivalent of an SQL `join` across the event streams, only matching when the `web:pageview` event and the `email:received` event are related in a specific way. *KRL event expressions are like SQL for the events in a PEN.*

Cloud-based identity context—Each PEN has an independent identity and the rulesets that operate within it operate specifically for that identity. For example, KRL has variables with values that persist from invocation to invocation of a given ruleset. These persistent variables are entity specific—that is, the values stored are for the entity who owns the PEN. This is quite powerful since it means that most applications don't need an external data store. But more importantly, we can use this entity-specific persistence to maintain online context for the owner of the PEN including links to external data or services.

APIs look like libraries—KRL is designed to ease the burden of working with Web-based APIs and their attendant protocols (e.g. OAuth). APIs form the libraries of the cloud OS. KRL makes it easy to bind multiple APIs together in a ruleset. KRL provides primitives for handling the predominant serialization standards like JSON, XML, and RSS, as well as providing modules for popular APIs and the ability for programmers to create and share modules for other APIs.

At present the only way to create programs in a PEN is using KRL. That needn't be the case. In the future, other programming languages might be used. They would need frameworks to understand events and the PEN's persistence model. For now, however, it's best to think of KRL as the C of the PEN; *KRL is how personal event networks are programmed.*

Services

Just as a computer operating system provides common services—like printing—that any program running on the OS can use, a cloud OS runs services for rulesets running in the personal event network. The first of these services we have developed is for notifications (http://www.windley.com/archives/2011/12/notifications_in_a_personal_event_networks.shtml) .

We have defined a draft Notification Event protocol

(http://developer.kynetx.com/display/docs/Notification+Event+Protocol) to show how notifications messages will be handled in a personal event network. The standard allows developers to write notification services. PENs will include a notification service by default, but the user can replace or augment the default service handler by adding rulesets that respond to notification events. In effect, the Notification Event protocol is the printer interface for a personal event network.

With a notification service in the PEN, KRL developers don't have to build notifications into their rulesets or worry about meeting the user's demands with regard to how they want to be notified. This makes writing rulesets easier because common operations and services can be handled by the COS.

In addition to notifications, we anticipate common services around such things as contacts, calendars, and to-do lists, and other key aspects of a user's personal data.

Unprecedented Power

Event-driven programming models are powerful because of several important, fundamental aspects (http://msdn.microsoft.com/en-us/library/dd129913.aspx) :

- **Receiver-Driven Flow Control**—Once an event generator sends an event notification, its role in determining what happens to that event is over. Downstream event processors may ignore the event, may handle it as appropriate for their domain, or may propagate it to other processors. A single event can induce multiple downstream activities as it and its effects propagate through the event-processing network.

  Unlike demand-driven interactions, event notifications do not include specific processing instructions. For example, if my phone receives a call, it can send the `phone:inbound_call()` event and the stereo system can interpret that as "turn down the volume". In a demand-driven architecture, my phone would send the `turn_down_volume()` instruction to the stereo instead.
- **Higher Decoupling**—Compared with other system architecture styles, event-processing systems exhibit higher decoupling along several important axes:
  1. Because event notifications do not contain processing instructions, destination information, and other details about how an event should be processed, the schema of the event is simple and flexible allowing less coordination between event generator and event processor. This makes events a convenient and legitimate means of creating lightweight protocols (http://www.windley.com/archives/2012/03/protocols_and_metaprotocols_what_is_a_personal_event_network.shtml) .
  2. Event generators do not necessarily need to know what processors are interested in the event. The event generator sends notification of the event to the event channel and, consequently, the associated event network, not to a specific processor.
  3. System components can be added or removed with less coordination in the overall system. Other components that want to respond in their own way to an `phone:inbound_call()` event can join the network without the event generator or any existing event handlers being affected. Similarly, components can be removed without the event generator and other event handlers being updated. This allows functionality to be layered.
- **Near Real-Time Propagation**—Event processing systems work in real-time in contrast with batch-oriented, fixed-schedule architectures. Events propagate through the network of event processors soon after they happen and can further affect how those processors will interpret and react to future events from the same or different event generators. Personal event networks thus allow a more natural way to design and create real-time information systems. As more and more information online gains a real-time component, this kind of processing becomes more and more important.

Personal clouds, running a cloud OS, with location-independent, semantically correct access to personal data from around the Web, and running applications that interact with other online services for the owner's benefit promise to usher in a new Web of unprecedented power and

convenience. This isn't possible in systems that merely respond to user interaction from a browser or an app as is done in current client-server systems. For this future to occur, people need access to systems that operate 24/7 and see things—events—as they happen. That's the power of the event-based programming model embodied in personal event networks and made possible with KRL.

Federating Personal Clouds

One of the most important aspects of personal clouds, as we envision them in this white paper , is their ability to federate. Without federation, personal clouds are as interesting as a computer without a network connection. *Federation* is a fancy word to describe what comes naturally to people: operating in a collective manner according to conventions or standards. In human terms, we call this being social. Neighborhoods, cities, clubs, and even supply chains and organized markets are federations of a sort.

In computer terms, *federation* usually refers to the interoperability of something: a network, a chat system, and so on. The Internet, for example, is a federation of networks that interoperate. One of the trends that seems inexorably at play in computer systems is the move from centralized solutions to distributed ones. We find it easier to build centralized systems first, but eventually we figure out how to decentralize them because that leads to not only better scaling and performance, but also greatly increased flexibility. Federation is the key idea that allows loose collections of decentralized systems to work in concert in pursuit of a common goal. *Federation* is organization.

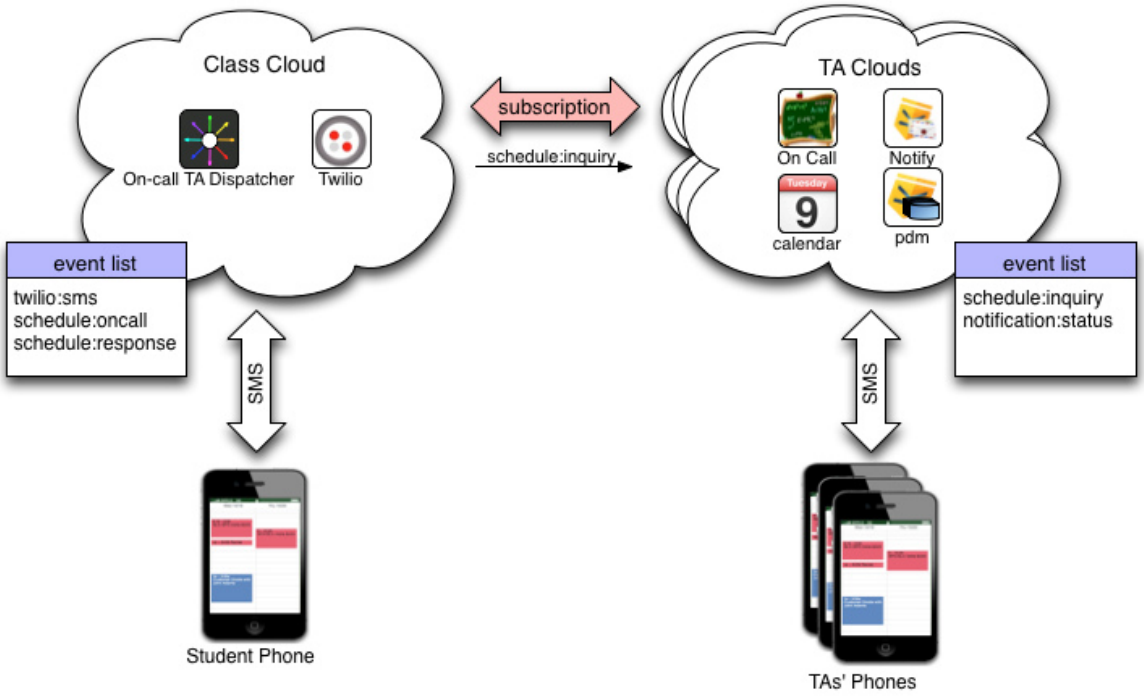
The programming model we discussed earlier is specifically designed to allow federation so that multiple personal clouds can participate in a collective solution to a problem. The magic that makes this possible is already built into the PEN: events, salience, and channels.

Personal event networks federate by subscribing to events from other PENs. Event channels play a key role in event subscription. When one PEN wants to subscribe to the events of another, it supplies an event channel and any other information that the publisher needs to complete the subscription. KRL has built-in features (http://www.windley.com/archives/2012/03/sending_events_in_parallel.shtml) for sending events to subscribers.

An Example: On Call Teaching Assistants

To illustrate the use of event subscription for federation between personal clouds, consider the following problem. Phil has two teaching assistants (TAs) for a class he teaches at BYU (http://classes.windley.com/462/) . (There are some large classes that have dozens.) The TAs are usually in the building, but not sitting at the TA cubicle. They are happy to be "on call" and answer student questions almost any time if they don't have to sit in the cubicle away from their usual desk. The solution is an application that allows students to text a single "class number" and request a meeting. Normally we might build that application as a stand-alone Web site, but personal clouds provide an opportunity to "think differently" about the solution.

The following diagram shows how we approach the problem. The system is made up of multiple, federated personal clouds. There is one for the class and one for each TA or instructor who wants to participate.



TA clouds subscribe to events from the class cloud

The PEN in the class' personal cloud contains a ruleset called the "On-Call TA Dispatcher" (`dispatcher`). As its name implies, this ruleset listens for incoming text messages and dispatches the requests to any subscribing TAs by raising a `schedule:inquiry` event. The TA clouds each have an "On Call" ruleset that is listening for that event.

Note: Don't let the term *personal,* as applied to the class' cloud throw you off. We use *personal* to refer to the idea that each event network is being operated on behalf of a specific entity—in this case, the class. We think *personal event network* sounds much better than *entity-specific event network*.

The scenario is straightforward:

1. The student sends a text to the class phone number requesting an appointment.
2. The `dispatcher` sees the event caused by the incoming text message and dispatches the `schedule:inquiry` event to any subscribing PENs.
3. The `oncall` ruleset in the TA's PEN, responds to `schedule:inquiry` events by checking the TA's calendar.
4. If the calendar indicates that the TA is currently on call, then the `oncall` ruleset raises a `notification:status` event inside the PEN.
5. The `notify` ruleset sees the `notification:status` event and sends a text to the TA indicating a student wants to meet.
6. If the TA is able to meet, she responds to the text.

Note that more than one TA might see the on-call request. Any TA who receives a notification and is available to meet merely responds to the text and the student is notified that help is on the way. If no TA is on call, the `dispatcher` ruleset tells the student that no one is on call right now.

Event Subscription

Let's explore the details of event subscription by looking at a little KRL. Subscribing to events requires that the subscriber (i.e. the TAs) provide an event channel and other information to the

event publisher (i.e. the Class). Event channels have an identifier, called the *event channel identifier* (ECI) that uniquely identifies the channel. The ECI is sufficient information to send an event from one PEN to another. To keep things simple, imagine the ECI and other information are stored in an array like so:

```
teaching_assistants =
  [{"name":"Phil",
    "phone":"801362XXXX",
    "eci":"072a3730-2e9a-012f-d2da-00163e411455",
    "calendar":"https://www.google.com/calendar/..."
   },
   {"name":"John",
    "phone":"801602XXXX",
    "eci":"fc435280-2b60-012f-cfeb-00163e411455",
    "calendar":"https://www.google.com/calendar/..."
   }
   ...
  ];
```

Given this subscriber list, the rule that dispatches events to the teaching assistants is fairly easy to write. The rule is selected when the PEN sees a `schedule:inquiry` event:

```
rule dispatch {
  select when schedule inquiry
    foreach teaching_assistants setting (ta)
      event:send(ta,"schedule","inquiry")
          with attrs = {"from" : event:attr("From"),
                        "message": event:attr("Body"),
                        "code": math:random(99);
                        };
      }
      always {
        raise explicit event subscribers_notified on final
      }
}
```

Notice that this rule loops over the `teaching_assistants` subscriber list using a `foreach` and uses the action `event:send()` to send the `schedule:inquiry` event to each TA in the list. When the rule is complete (note the `on final` guard condition in the postlude), it raises an event called `subscribers_notified` so that any final processing can be done. The `event:send()` action raises events to subscribing networks in parallel.

Federation through Subscription

The preceding example shows how multiple personal clouds, running an event-based cloud OS can federate to accomplish a simple task for their owners. The class has a personal cloud that is running rulesets to manage the class' business. The TAs each have clouds running rulesets to manage their business. These networks are owned and managed by different people. And yet, through subscription, the student finds an on-call TA to help them with their problem. Federation has allowed these various clouds to cooperate in solving a problem.

There are a few things to point out:

- None of the rulesets know about the others. They are connected by through salience in a loosely

coupled manner.

- The behavior of the rulesets isn't specialized to an individual. The rulesets are general purpose. All the personal data is retrieved from the personal data manager installed in the personal cloud.
- The subscriptions are made using event channels that create a one-to-one link between networks. This protects against SPAM and other communications abuses. The relationship can be revoked as easily as it is created without affecting other relationships. The subscriber (TA) controls what events it sees by managing event channels.
- Any network that wants to see the `schedule:inquiry` events from the class personal event network *must* subscribe to them. The publisher (Class) controls who sees the events by managing the subscription list.
- With the exception of the `oncall` ruleset, the TAs would likely have different rulesets installed in their PEN. For example, each TA could have a different `notify` ruleset installed as long as they each understood the `notification` event.

Federation arises from the properties of the programming model that we discussed earlier. Most important among these are event channels and salience. Event channels provide a way for PENs to communicate securely and privately. Salience ensures that events are routed to the rules that need to see them regardless of their source.

Through event subscription, personal clouds can cooperate to automate interactions that in other circumstances would necessarily be driven by the actions of their users. The On-Call TA use case supplies several examples:

- The TAs don't have to check anything to see if students are waiting. The system merely notifies them when they are.
- Personal data like the TA's own calendar modifies the overall behavior of the system.
- TAs can subscribe or unsubscribe as they start work or leave employment without the students or other TAs needing to change anything.
- TAs are notified in the manner they choose, based on their personal preferences and the rulesets they've activated.
- The TAs are represented in the interaction by a system they control. This may not be important in this scenario, but could matter a great deal in other scenarios, like those involving commerce.

Federation turns personal clouds into automated assistants. A personal cloud becomes a personal valet that thinks about you—*your needs, your quirks, your life*. Rather than having to get involved in all the gritty details of how things get done, you're in a position of just making go—no go decisions, like the TAs in the example above. As a new category of personal cloud apps are written to take advantage of federated personal clouds, this new form of cloud-based personal assistant will soon become as indispensable as a smart phone is today.

Moving Beyond Federation

As more and more of our interactions move online, we increasingly have need of an online place that operates for us. Personal clouds must become more than appliances to achieve their real potential. While appliances provide value, they can't anticipate every need. They aren't flexible enough.

This paper has outlined a vision of personal clouds as general purpose virtual computers. Making that vision real requires an operating system so that developers have a framework to work within. Operating systems provide a core set of services around identity and data as well as a programming model. We have set forth our vision for how those services will work.

Like smart phones, federated personal cloud networks will fundamentally change how businesses and customers interact. While direct federation is a powerful tool, there are many online interactions that require participation between parties who are unfamiliar with each other and don't trust each other. This is the subject of the next paper in this series: *The Personal*

Channel: Connecting Customers and Companies Like Never Before.

Finding Out More

You can discover more information about the concepts and technologies in this series from a variety of sources including Project VRM (http://blogs.law.harvard.edu/vrm/) , Respect Network (http://respectnetwork.com/) , Kynetx (http://www.kynetx.com) , and the blogs of the series authors (listed in the biographies below). We also point you at Doc Searls' book The Intention Economy (http://www.amazon.com/The-Intention-Economy-Customers-Charge/dp/1422158527) and Phil Windley's book The Live Web (http://www.amazon.com/exec/obidos/ASIN/1133686680/windleyofente-20) .

If you're interested in creating personal clouds, the Kinetic Rules Engine is open source (https://github.com/kre/Kinetic-Rules-Engine/) . However, the easiest way to get started is using the online service provided by Kynetx. You can try out personal clouds and the KRL programming model for free by creating an account at Kynetx (http://www.kynetx.com) . Kynetx accounts are free and you can develop multiple applications and run them without charge for non-commercial use. Examples and documentation (http://developers.kynetx.com) are available online.

Rights

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License (http://creativecommons.org/licenses/by-nc-sa/3.0/) .

Live Web Series Authors

Craig Burton is a Distinguished Analyst for KuppingerCole. As the creator of breakthrough concepts like NetWare Open Systems, The Network Services Model, metadirectory, and The Internet Services Model, Craig Burton is one of the leading visionaries and analysts in the computing industry. He is a frequently consulted expert on new technologies and the process of making software infrastructure ubiquitous.

Craig's visionary understanding of new technologies and his familiarity with the needs of technology users led him to identify the market for a metadirectory product and allowed him to articulate the essential attributes of such a product. Craig invented the conceptual basis for metadirectory over 20 years ago while leading Novell, Inc. to success.

Craig was one of the founding members of Novell, where he served as senior vice president of corporate marketing and development. During his eight years with Novell, Craig's unique market strategies (which included the development of concepts such as file server technology, hardware independence, fault tolerance, Universal NetWare Architecture and NetWare Open Systems) resulted not only in market leadership for Novell, but also served to accelerate the movement toward transparent multivendor computing.

When he co-founded The Burton Group in 1989, Craig began to study the concept of metadirectory in earnest. As CEO, president and principal analyst of The Burton Group, Craig provided independent market analysis on network computing technology and industry trends.

Craig's contributions to computing have earned him recognition as one of the industry's most influential analysts. Since 1997, Craig Burton has been an independent analyst consulting with many industry vendors and concentrating on the Internet Services Model and the paradox of selling software infrastructure. Craig blogs at http://www.craigburton.com/ (http://www.craigburton.com/) .

Scott David is a partner working with the electronic commerce, tax, and intellectual property practices at K&L Gates, LLP. He provides advice to firm clients on issues of international, federal, state and local taxation; intellectual property licensing and structuring; compliance with federal and state privacy and data security laws; structuring of online contracts, terms of use, privacy policies and electronic payment and tax administration systems; corporate, partnership and limited liability company organization and affiliation structuring; technology development and transfer; participation in standards setting organizations; and non–profit and tax–exempt status and related issues. He regularly counsels the firm's intellectual property, high technology, telecommunications, on–line commerce, power generation, construction, retail, manufacturing, service sector, health care, governmental, financial sector and other clients.

Drummond Reed is co-founder and Chairman of Respect Network Corporation (RNC) and Managing Director of the Respect Network. He is co-author with Scott David, Joe Johnston, and Marc Coluccio of the Respect Trust Framework upon which RNC's Connect.Me (http://connect.me) reputation network is based. Drummond has also served as co-chair of the OASIS XDI Technical Committee since 2004.

Prior to RNC, Drummond was Executive Director of two industry foundations: the Information Card Foundation and the Open Identity Exchange. He has also served as a founding board member of the OpenID Foundation, ISTPA, XDI.org, and Identity Commons. In 2002 he was a recipient of the Digital Identity Pioneer Award from Digital ID World. Drummond blogs on digital identity, personal data, personal clouds, and trust frameworks at http://equalsdrummond.name (http://equalsdrummond.name) .

Doc Searls In *The World is Flat*, Thomas L. Friedman calls Doc Searls "one of the most respected technology writers in America." Searches for Doc on Google tend to bring up piles of results. Doc is Senior Editor of Linux Journal, the premier Linux monthly and one of the world's leading technology magazines. He is also co-author of The Cluetrain Manifesto (http://www.amazon.com/The-Cluetrain-Manifesto-Anniversary-Edition/dp/0465024092/) , a book that was Amazon's #1 sales & marketing bestseller for thirteen months ("Cluetrain" now appears in more than 5,000 books), and author of The Intention Economy: When Customers Take Charge (http://www.amazon.com/The-Intention-Economy-Customers-Charge/dp/1422158527/) from Harvard Business Review Press.

Doc serves as a fellow with the Center for Information Technology and Society at the University of California, Santa Barbara and is an alumnus Fellow with the Berkman Center for Internet and Society at Harvard University, where he continues to lead ProjectVRM, which has the immodest ambition of liberating customers from entrapment in vendor silos and improving markets by creating a productive balance of power in relationships between supply and demand. Doc is a pioneering and highly quoted blogger. J.D. Lasica, author of Darknet, calls Doc "one of the deep thinkers in the blog movement."

As a writer, Doc's byline has appeared in Harvard Business Review, OMNI, Wired, PC Magazine, The Standard, The Sun, Upside, Release 1.0, Wired, The Globe & Mail and many other publications. He is @dsearls on Twitter.

Phillip J. Windley is the Founder and Chief Technology Officer of Kynetx. Kynetx is a personal cloud vendor, providing the underlying technology for creating, programming, and using personal event networks using KRL and semantic data interchange via XDI. He is also an Adjunct Professor of Computer Science at Brigham Young University where he teaches courses on reputation, digital identity, large-scale system design, and programming languages. Phil writes the popular Technometria blog (http://www.windley.com) and is a frequent contributor to various technical publications. He is also the author of the books The Live Web (http://www.amazon.com/exec/obidos/ASIN/1133686680/windleyofente-20) published by Course Technology in 2011 and Digital Identity (http://www.amazon.com/dp/0596008783/windleyofente-20) published by O'Reilly Media in 2005.

Prior to joining BYU, Phil spent two years as the Chief Information Officer (CIO) for the State of Utah, serving on Governor Mike Leavitt's Cabinet and as a member of his Senior Staff. Before entering public service, Phil was Vice President for Product Development and Operations at Excite@Home. He was the Founder and Chief Technology Officer (CTO) of iMALL, Inc. an early creator of electronic commerce tools. Phil serves on the Boards of Directors and Advisory Boards for several high-tech companies. Phil received his Ph.D. in Computer Science from Univ. of California, Davis in 1990.