Designing PWC’s remote monitoring system[1]

Peak Web Consulting’s management toolset will include a tremendous amount of reporting, analysis and data modeling systems, all designed to give customers maximum visibility into what their network is doing and allow PWC to run it efficiently. Our tools will be a huge part of our value proposition.

But the tools are hampered by a poor infrastructure for data collection: it’s all done now from the data center in San Jose over a distance using the public internet. This has a long list of downsides:

- We miss polls due to packet loss
- We cannot do anything in private RFC1918 space
- Customers are concerned about the security implications of external monitoring
- Our systems are completely disjoint, sharing nothing (neither configuration nor data). Example: happydog[2] and sat-monitor[3] collect the same data but never share it.

This will not scale – and *even now* is a maintenance nightmare.

This is a proposal for a remote-monitoring architecture for PWC, the first half of our toolkit. It will provide for integrated configuration of all customer devices from the central location, secure deployment of that configuration to remote monitoring devices, remote data collection, and secure, queued replication of polled data back to the central location.

Once back at the mothership, the data gets broken out to the various systems for processing in a system-dependent way. With an integrated configuration toolset that the second-stage tools can use, and data already gathered, it will be a ripe ground for making products that customers will love.

The two parts – collection and reporting – will share a common data-queuing core and directory structure that allows for easy cooperation between separate “departments”.

1 Overall design requirements

Rather than start with how the system will work, it’s important to see which needs it’s designed to address. “Just collect some data” is much too imprecise, and it’s important that we understand the complexity of what’s in front of us.

[1] This is an ad hoc document for internal use only, and has had only the most rudimentary proofreading performed. It’s never going to be published, and the demands of time have not allowed the kind of writing care usually associated with my work – Steve

[2] Legacy monitoring system, currently the customer-facing graphical interface to display bandwidth and traffic. Originally designed by jpapen while he was at Yahoo!, it’s creaking under the strain of being used for things it wasn’t really designed for.

[3] Separate monitoring system that grabs bandwidth data at 60-second intervals and lets us check near realtime link saturation.

1.1 Secure

This is at the front of the list: we must be secure not only from outside parties (which we assume will be straightforward), but from our own customers: in many cases, even the mere presence of a PWC/customer relationship is confidential information.

It's not so much that we're worried about outright industrial espionage by one customer towards another (though this gets more likely as we grow), but the inadvertent leak provided for by a software problem that turns into a terrible PR problem. It's remarkably easy to code your software to always check for a certain customer number, but simply mess up and include *all* customer data by mistake. We're always just one SQL injection away from giving away everything.

Security is such an important issue that we'll address this in much more detail later, but we have to build our systems such that a customer's data center guy can *walk off* with our box and not learn anything about customers other than his own, and that if the bad guy is root *on our box*, he can't really hurt us at the mothership. Monitoring pods are in a hostile environment.

1.2 Distributed

We simply cannot monitor everything properly from the PWC data center in San Jose: even if one discards the losses over the public internet, there are simply too many customers with private RFC1918 space to ever seriously consider distant polling.

There are several mechanisms that allow for remote polling of private space:

1) **A VPN-like solution** that gives a private tunnel from the mothership to the customer's network, but any solution that involves actual routing to the private space means that we run into address conflicts all the time. How many customers use 192.168.x.x ?

   There is *no way* we can tell customers they can't use an IP range because one of our other customers is using it – You should have laughed out loud at this notion.

2) **SSH remote commands**: one-off commands can be run from the mothership proxied through a remote jump box, and this is not a terrible solution for small amounts of data. But this only tunnels TCP, not UDP, and it's not really suitable for bulk data.

3) **Application-level proxies**: SNMP v3 provides for "contexts", which allow for one SNMP server to do what amounts to proxying to another system. Not all the devices we wish to monitor support SNMPv3, and it's much more work to configure. None of our software now supports SNMPv3 except Cacti, and this has not been proven.

The only circumstances where remote polling makes sense is when the number of devices is small and the customer relationship is minimal (say, a new customer where we need to set up simple Happydog-esque monitoring in 2 hours, like we did with new customer SyncCast). For customers who have actually retained PWC for work, they probably warrant a data-collection box.

1.3 Autonomous

The remote collection pods must have enough local intelligence to continue as much of their work as possible even when disconnected from the mothership. There are limits to this, but the pods should certainly not require a connection to the main office every time it's ready to start a five-minute polling run.

To have this realtime phone-home requirement would put the distributed solution in the same position as doing remote polling: we're at the mercy of Limelight or every other provider between A and Z.

This can be achieved either by replicating a configuration store to the collection pods, or by a phone-home connection that simply caches the last set of instructions, and repeats them when the link is lost ("Well, I can't ask mom for my to-do list, so I'll do the same thing I did 5 minutes ago"). Either solution would temporarily isolate the pod from changes made at the mothership, but they would continue with the best information they had at the time.

1.4 Partitioned

We have to assume that these remote monitoring pods are in an entirely hostile environment, exposed to the internet and to customers, and that one of them may walk out the door and be analyzed by a skilled network engineer with knowledge of our customer base.

1. The box cannot have data belong to customers not being monitored: even though it might be easier to replicate the entire configuration store.
2. We must separate our customer access credentials: anybody getting his hands on the box will learn the SNMP community strings we use for monitoring customer A, but this should not give them any leverage for customer B (Access Control Lists notwithstanding). We will use separate credentials for each customer, but because they are stored in the directory, it will be no problem for the tools to deal with this.

Partitioning is *so important* that we address this further in a later section.

We have no solution that does not involve rigorous customer partitioning.

1.5 Low-footprint

We expect to offer an enormous number of analysis, reporting, and modeling applications for customers, and this means a lot of data collection, but we wish to build the smarts only at the mothership. By creating the monitoring pods without any knowledge of why they are collecting the data, we don't have to update the collection software as much when the applications change.

This can't be entirely avoided – we will have odd cases from time to time – but the more we can make the operation *driven by the configuration store* and not the software, the easier it will be to deploy. We'd love for an engineer to install a machine with standard Fedora Core installation, and overlay the monitoring system and configure it by pointing to our configuration store, and be done with it. Limited hand tuning of each location.

The less software that's found on the pods, the less PWC intellectual property is at risk in the hostile environment with the customer. They may walk off with the box (and the software), but they don't get the real secret sauce back at the mothership.

Finally, this approach allows for much lighter weight boxes at the remote sites. Being able to live with cheaper boxes collecting data – even at the cost of bigger ones at the office – is a solution that scales well.

I expect we'll be able to purchase a moderate 1U Dell box, with plenty of RAM and disk, plus onsite support, for in the $1000 range – this gives PWC a lot of room to charge for the software portion. Lightweight box = earlier ROI for profit on tools.

1.6 Integrated

The state of PWC monitoring systems is a mess: every system is completely independent, and the same devices must be configured in a dozen different places. Only one person on staff (me) has any idea how to do this, and very little of it is documented. It now takes about two hours to set up a new customer for monitoring – much of this is spent on duplicate entry and wandering all over the data center systems.

This whole process was made somewhat easier some time ago by my adoption of DNS as the hostname store for the `*.cust` domains[4]: the `dev-query` program performs a zone transfer if the customer's zone, and this defines the devices we know about and displays it all in a web page. Some systems use it, others do not, but at least we have a centralized store for hostname to IP.

This has been an ugly over-the-hill-and-through-the-woods process every time we do this, and this is what led to the creation of the dev-coverage system: though the engineers all use it as a quick reference to customer devices, the driving purpose was to insure that we actually included each device in each system.

"Uh oh, we added a device but forgot to include it in Happydog" drove this program, though it badly needs to be extended for sat-monitor and all the other things we do around here.

What we need is a centralized store for *everything* related to setting up any of our soon-to-be many monitoring systems. All the systems will share the obvious common information (hostname, IP address, description), but each system will allow its own configuration bits. "Don't include this device in happydog" and "This is the RANCID information" all gets associated in a common place with a single interface.

This means modification of existing systems to support this: RANCID (for instance) will have its configuration files generates automatically from the config store, so adding a new device via the central web GUI adds to RANCID without a trip over to apu.

1.7 Customer Partitioned

This is a crucial element of security, and I ***cannot emphasize enough*** how important this factor is. You ***must*** own this factor in your heart and in your soul.

We certainly plan to use due care while coding in order to protect the data from inadvertent disclosure, pay attention, and the like, but you cannot ever fully avoid bugs. What you have to do is set up *strategic* mechanism – not just *tactical* programming steps – that protect the data.

Example #1**: Cacti**

Let's say that one Cacti instance can reasonably support 100 devices. Cacti supports multiple graph trees and multiple users, and you can assign permissions to each part of the tree. It's straightforward to allow **oversee** to see one part, **myspace** another, and **pwh** yet one more. This works now.

---

[4];`*.myspace,cust` `*.facebook.cust`; `*.jupiter.cust`; etc.

But you are just one configuration mistake, one Cacti bug, one misunderstanding of the subtleties of Cacti permissions, from giving away one customer to another. ***This has happened*** in the past, though (luckily) with no ill effects.

What if, instead, we created ***separate Cacti instances:***

- Separate directory holding the code + config
- Separate database (`cacti-myspace` *versus* `cacti-facebook`)
- Separate database user (each with no permissions on the other db)
- Separate access URL

This does require a bit more code to support the routine polling – you script the MySpace polling, then facebook, then the next, etc., all in the 5-min polling interval, but it provides wonderful customer partitioning. A bug in cacti? SQL injection? Nothing affects another customer.

Not only does this make for much better security, but it allows for much easier configuration of the Cacti instance: we assume that there are only two users – pwc (for admin) and the customer (for viewing graphs), and graph viewing is configured to be allowed for anonymous users.

But because we use *Apache authentication* to even get to the URL, we have avoided the issue of Cacti permissions entirely. This is an enormous win for security, and as a free bonus makes the administration of Cacti much easier.

This is *systematic* customer partitioning that lets us look a customer in the eye.

Example #2: the *.cust zone

We maintain DNS for customer-monitored devices in `*.myspace.cust`, `*.jupiter.cust`, and so on. This insulates us from customers messing up their own DNS or having to wait for them to get around to it after we bring up new equipment (we can do this in very short order). It's been a Godsend not to have to use IP addresses to get to customer equipment.

But this is not the only reason we do this: it's was driven by customer partitioning security. For any tool that addresses customer equipment (say, `rancid` or `dev-query`) the parameter provided to the tool undergoes the usual security sanitizing. But it's then appended with the customer domain, and if the resultant name is not found in DNS, then it's a bad name.

This has the dual effect of not only insuring that names not in our DNS do not exist, but it forestalls any shenanigans that attempt to see *other* customers. If a staffer at MySpace ran

```
    dev-query.cgi?device=br1.lax1.facebook.cust
```

and actually got somebody else's data, it would be very bad.

But this DNS partitioning scheme completely shuts that down. It's just not possible to get other customer data no matter how much you game the inputs. We do minor sanitization of the device= inputs, but only for bogus characters. We completely rely on DNS to keep the user out of another customer's data.

The `.cust` domain is PWC's friend even if you don't realize it.

If one customer accidentally sees other customer data, he's going to assume that others can see his, and it's hard to overemphasize how powerful this blunder would be. For some time, customers who used the `custom_graph.cgi` facility of Happydog ***saw all customer devices***, and it is only by the grace of God that Ken Kellar did not run across this: It would have taken him 2 seconds to realize what he was looking at.

Using a good partitioning scheme certainly doesn't mean that we can be sloppy elsewhere, but since we have *systematic* security precautions, and not just "we're really careful", we can look customers in the eye and tell them that we have gone above and beyond to protect the sanctity of their information. And mean it.

2 The Directory

The configuration store for this entire system is to be an LDAP directory – this is *exactly* what LDAP was designed to do, and is far superior to the other alternatives, such as SQL or text files. Microsoft's Active Directory is implements LDAP, and I can't see how it could ever do such with SQL.

This is going to be the hardest sell in this entire proposal, because LDAP is not well understood inside of PWC and may be seen as an attempt to use something cool, but more than a year of research on using it for this purpose make it clear that it's the right answer, and that the other answers suck.

We'll consider this in the context of the problems it's meant to solve, and with many comparisons and contrasts to the use of SQL.

2.1 What is LDAP?

LDAP is an access protocol on top of a persistent data store, and we're treating it as a database. It's like SQL in many ways – including the important ways – (each record has multiple fields, indexes, queries, etc.). It's commonly used for phone-book type application and is widely deployed for centralized access control. LDAP has been around for a long time and has well-settled standards and toolkits.

Addressing used by LDAP is *very* intimidating, and this is a substantial barrier to adoption. By way of example, this is the name of a potential device in one customer's hierarchy:

```
cn=ni-a.myspace.cust,ou=devices,ou=myspace,ou=cust,dc=peakweb
```

It actually reads right to left, but since we don't want to get bogged down on the notation, we're going to instead invent a UNIX path-like mechanism to show the same thing. This has nothing to do with the filesystem, is not any kind of common notation elsewhere – it's just using a familiar way to show what the ugly notation represents:

```
/peakweb/cust/myspace/devices/ni-a.myspace.cust
```

This represents one object[5] (a row) inside a container (a table), and this contains the obvious information about a device: the hostname, IP, SNMP credentials, device type, and flags that indicate whether this participates in the various systems (happydog=yes, rancid=no, etc.).

There are several things that make LDAP so attractive here.

[5] We use the term "object", but LDAP is not really an OO database in the sense it's normally used.

2.1.1 LDAP is hierarchical.

We can not only easily populate the tree of MySpace devices, but do so on a per-data-center basis. For small customers this won't matter so much, but MySpace could have *hundreds* of devices. We don't want to make all users look at all devices all the time – it's just too much overload.

```
/peakweb/cust/myspace/devices/
                        els1/idf2ds1a.els1.myspace.cust
                        els1/idf2ds1b.els1.myspace.cust
                        ...
                        chi1/br1.chi1.myspace.cust
                        chi1/br2.chi1.myspace.cust
                        ...
                        ash1/br1.ash1.myspace.cust
/peakweb/cust/facebook/devices/
                        sctm/br01.sctm.facebook.cust
                        sctm/br02.sctm.facebook.cust
                        ...
                        sf2p/bf01.sf2p.facebook.cust
                        ...
```

The hierarchy is supported *by the directory itself*, and it works automatically for any kind of object. One can do this with SQL, but it requires additional application-level support and extra linking tables to make it work. One level of hierarchy is merely tedious, but arbitrary tree depth is a horrible application nightmare with a relational database.

Querying the tree can be done on a hierarchy basis, so starting at a given node, we can get all nodes at that level, or at that level *or below*. Continuing our UNIX filesystem analogy, let's use the quasi-`find` command to locate some devices:

```
find /peakweb/cust/myspace/devices -type device -print
```
retrieves all MySpace devices

```
find /peakweb/cust/myspace/devices/els1 -type device -print
```
retrieves all MySpace devices just at the ELS1 data center

```
find /peakweb/cust/ -type devices -print
```
retrieves all customer devices

The results show that the individual devices are all at different levels, but accessing them one at a time is a "flat" operation – the consumer of the search need not be aware of tree depth. Iterating over a collection of 2 devices or 10,000 devices in the tree uses the same interface.

This works for any kind of organization (devices, users, whatever), and we'll certainly have a separate part of the tree for things like circuit organization: which device+interfaces are Level3, which are peering, which are part of the totals, etc. Moving things around in a tree is straightforward (say, moving the NetScalers from LAX1 to LAX3).

Each object has a type ("device", "user", etc.), and can live at any place in the tree. There's a tremendous amount of information that we'll need to store about our customer systems, and being able to organize it in a hierarchical manner is going to be really important.

2.1.2 LDAP supports multi-values attributes

SQL explicitly does not. The classic example here is *group membership* for mailing lists (such as we use for various support aliases here). Both approaches require a user table and a group-name table, but since LDAP allows more than one attribute with the same name, we can define (in a made-up format):

```
object /peakweb/cust/myspace
custName: MySpace, Inc.
custURL: www.myspace.com
supportingEngineer: steve@pwc
supportingEngineer: droisman@pwc
supportingEngineer: jeffrey@pwc
```

Multiple attributes are a completely natural part of the system. SQL requires a separate group membership table.

Queries[6] can be done this way:

```
find /peakweb/cust -type customer -where supportingEngineer=steve@pwc
```

this finds all customers where I'm on support (though the SQL query would not be any more difficult even though it requires the extra table).

The need for non-atomic attributes comes up *all the time*, but because SQL people don't have this available (and have to use linking tables), they simply don't think about them – the effort of having to create separate linking tables is often too much of a disincentive. No such problem in LDAP.

I have a lot of experience in another system that supports multi-valued attributes: the Pick OS. This is an operating system built around BASIC, and though the computer scientists among us turn up our noses, it was very popular (and very effective) for business applications back in the eighties and nineties. I worked on a large manufacturing system (Dataflow) for a customer, and I got a real appreciation for how helpful it was to *think* multivalued, and how much of a hindrance the relational model is for some classes of problem.

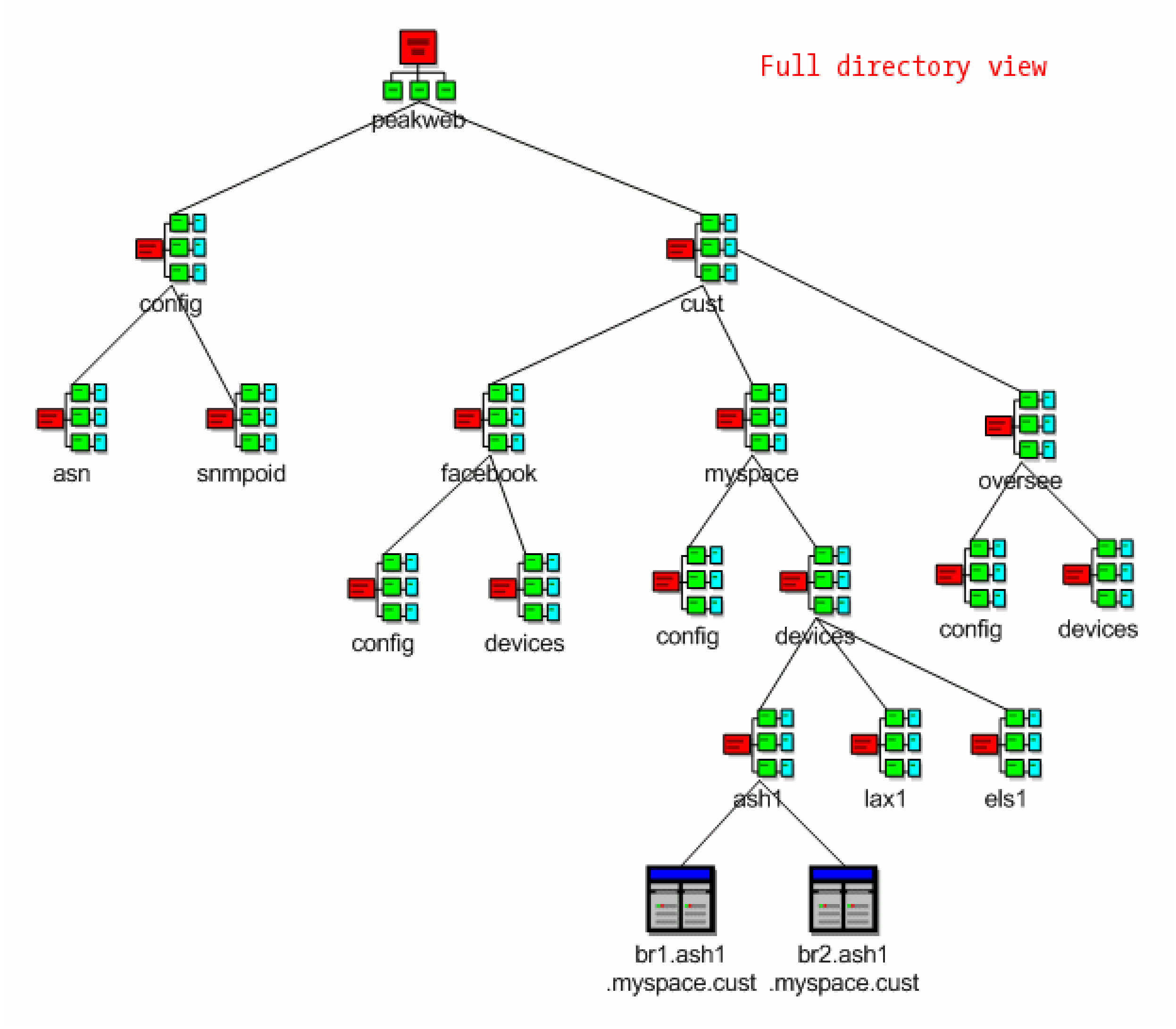
Those who think all problems have to be <u>relational</u> problems are not using the whole toolbox.

But so far these are merely conveniences, and the benefits may not outweigh the general unfamiliarity with LDAP. It's in customer partitioning and replication where LDAP is the big win.

2.1.3 LDAP is easily partionable

Like the UNIX filesystem, LDAP's directory is tree-structured, and (like UNIX), permissions can be applied at each node. Let's consider this partial tree:

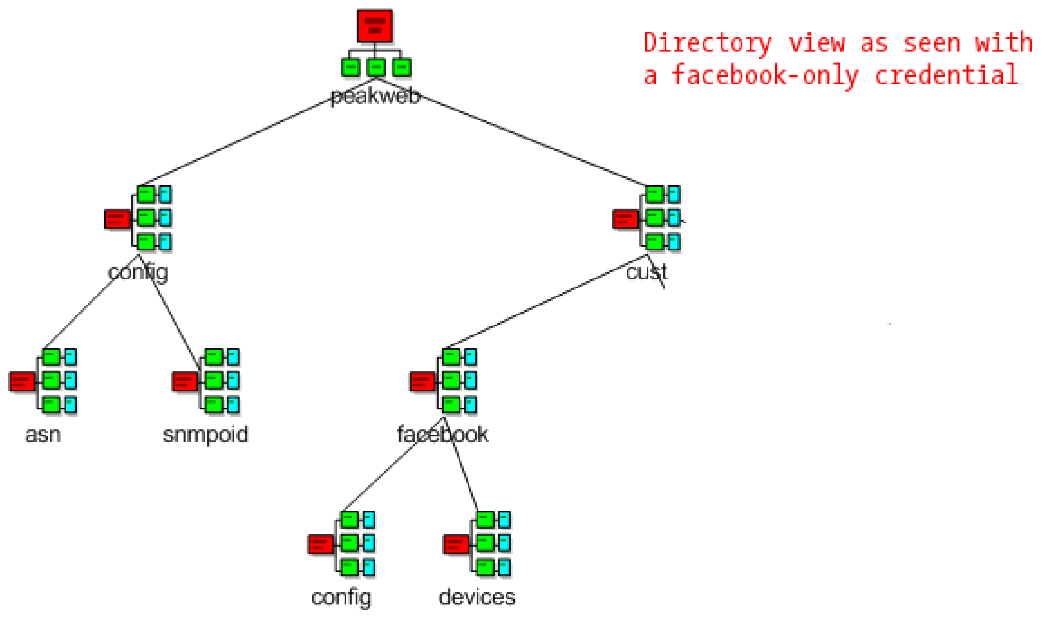
[6] Again, this is a made-up syntax simply to demonstrate how it work.



A user with the proper credentials sees the whole tree, and we'd certainly set up the **peakweb** user with those full rights to the whole shebang.

But the top of each customer tree would be restricted to users '**peakweb**' and that particular customer's user, and all of our tools via the presentation web interface would use *that customer's LDAP credential* to access the directory.

With these permissions, we effectively make invisible any part of the tree not accessible to them:



Unlike the UNIX filesystem analogy, where you can see the names of subdirectories you can't access, no such concept exists in LDAP: inaccessible parts of the tree **do not exist**. This means that all of our tools will query from the `/peakweb/cust/` node of the tree, asking for devices or data centers or whatever, and we rely on the permissions system to fetch only data for *that* customer.

```
find /peakweb/cust –type device –print
```

and they see only their devices, and not even a clue that there is anybody else in the tree.

No visible other-customer nodes, no permission errors for inaccessible items, no sign that this user can't see the whole tree. And the same query works whether it's the limited customer view as the full **`peakweb`** view. We can define users who can see all of a particular customer, but not the password/SNMP community fields as well.

Remember: we have oversimplified our directory by containing only 'devices', but in practice there will be many, many tables/objects: interface groups for billing, peering information, contact information, data center descriptions, and 100 other things as our tools get more and more sophisticated.

Objects can be centralized or spread out, and both are useful in our context.

Centralized: We'll have a `/peakweb/snmp/*` node that contains definitions for all SNMP OIDs we wish to maintain. It's much too much work to install and maintain the stock SNMP MIBs on every monitoring box we set up, plus the extra ones we're monitoring (NetScaler, obscure Cisco/Foundry), and we wish to add our own information anyway – like a notepad description and processing hints. We will live and die by SNMP, and we want very strong support for it.

All "`SNMPoid`" objects will live strictly in the `/peakweb/snmp/*` part of the tree, and all users have readonly access (but `peakweb` has the ability to write to it). This whole mechanism corresponds exactly to the use of a single "`SNMPoid`" table in SQL.

Decentralized: We'll support a "user" object that we use to describe a person and hold his password – this is how access credential checks are made. But including them all in the `/peakweb/users/*` hierarchy (which would correspond to a "`users`" table SQL) would not honor customer partitioning.

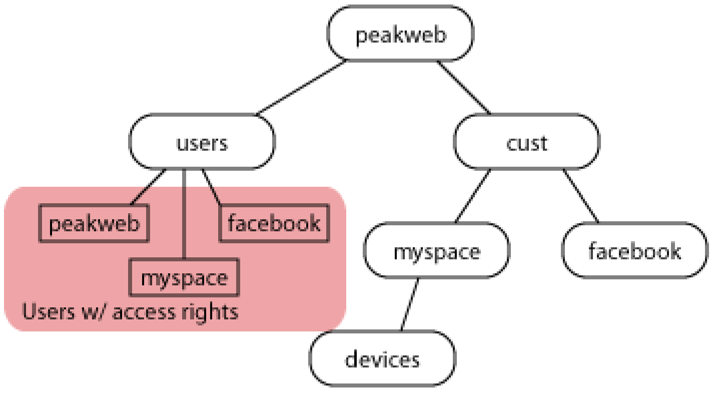


Figure 1 - Centralized user store

Much like the UNIX password file or MySQL users store, mere presense of a name doesn't grant access to particular objects, but any users who do have access receive it by dint of an entry in `/peakweb/users/*`. This means that replicating our store to a remote pod would have to somehow filter out users who should not belong on that pod (say, the user "`facebook`" should not appear on the MySpace pod).

But when we spread out the users, and keep them with their respective customer, this problem solves itself for free:

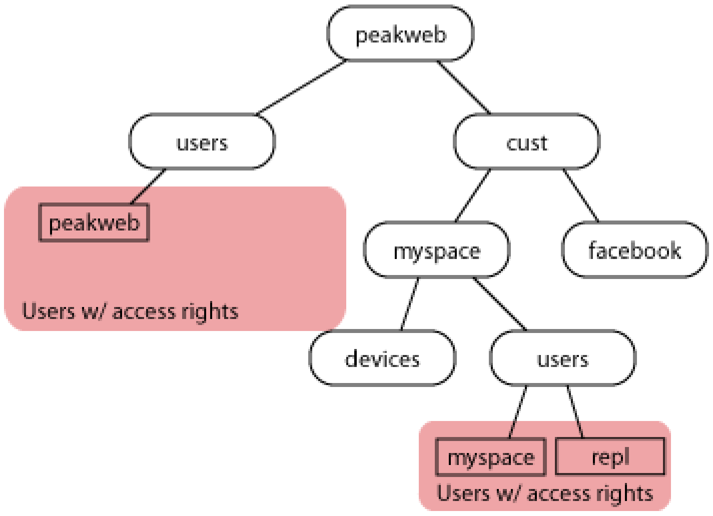


Figure 2 - Decentralized user store

The directory bind is done with the name of the user to bind as (which is the unit of permissions granting), and we can bind with either a `peakweb` user or that of a particular customer (the customer's replication user has mostly the same permissions as the customer user itself). Since the `/peakweb/cust/myspace/users/*` store replicates only to a MySpace pod, there is *nothing* about MySpace – even the access users – on other customer pods.

This will be much more difficult to achieve cleanly in SQL.

Though SQL has notions of per-table and per-column security, it does not have *per row* security, so one must either build in application-level security mechanisms, or set up separate tables or databases for each customer and forever give up cross-customer aggregation.

The `sat-monitor` tool uses separate tables to maintain customer partitioning, and it's *awful*.[7]

Now we turn to config replication over to the remote data-collection pods. We cannot have any part of one customer's information on another customer's pod, and we'll use the same mechanism for this as we do for partitioning for the presentation system.

2.1.4 LDAP Replicates on a partition basis.

We'll define customer-specific readonly replication users with rights to just the parts of the tree that matter, and the built-in LDAP replication mechanisms allow us to populate the directory on the pod with that information only.

SQL replication will not handle this case: it is all or nothing on a per-table basis, and we're faced with building an application-specific replication system (ugh), or making each customer's information in a separate database. Now we've (again) lost the ability to survey the whole cross-customer data store.

The replicas on the monitoring pods are strictly readonly, with all config changes being made at the master and pushed out to the pods. All the software on the pods looks to the directory, *exclusively*, for directions on what to gather. This includes device names, credentials, and which devices are part of each kind of system (Happydog=yes, rancid=no, etc.)

We do expect that data from the remote pods will find its way back into the directory, but it will strictly be done indirectly by application software on the main systems. For instance, the "last restart" time is a value that's easy to store back in a device's node, but the polling box that discovers this value doesn't update the directory. Instead, it queues the collected data back to the main office, where somebody else updates the directory.

This is a perfect fit for LDAP.

2.2 Where's the bulk data?

We're not sure, but probably not in LDAP.

We're using LDAP *only* for the configuration store, not the bulk time-series data. LDAP is optimized for read-mostly, and it's not really designed for the kinds of huge data sets such as happydog bandwidth data.

Instead, each application will choose its own format for storing what it likes.

- Happydog might use MySQL
- Netflow analysis might use Berkeley DB
- RANCID uses SubVersion/CVS

[7] Like many other tools around here, `sat-monitor` was written for a short-term *ad hoc* project, but took on a life of its own

but all are driven by the directory.

It may be possible that some part of the actual statistical data will be stored into LDAP itself (last time we reached the device, last reboot time, a mirror of the interface parameters, etc.), but this will depend on the nature of the data itself and of the designer of the particular application.

Note that the directory can also include the access credentials for the "other" system, so that (for instance) if Happydog is storing its data in MySQL, then the myspace part of the tree contains the MySQL username and password for the Happydog databases. Once our application has access to the directory, it can fully bootstrap itself for the rest of the access.

2.3 Programming with LDAP

… is no more difficult than SQL. Both require that you bind to the store with credentials, and queries are similar enough that a skilled perl programmer who knew how to do this with MySQL could learn the LDAP equivalents in short order.

The hard part is getting one's arms around a hierarchical, non-relational system, but this will be helped dramatically by there being sample code for our environment already existing, allowing other developers to just use a template to (say) "get all devices" or the like.

Writing the user-interface code to manage the config store should not really be any more difficult than the SQL version either, and in some respects will be much easier: knowing that the directory is managing the hierarchy and security aspects takes a tremendous burden off of the developer.

A developer with substantial experience with MySQL and perl will be a fine candidate to code to LDAP: we'll have excellent LDAP mentoring resources[8] available.

3 Remote Data-Collection Pods

The directory is maintained at the main office, but we must do monitoring from remote pods in order to reach RFC1918 space at customer data centers, as well as to insulate ourselves from outages[9].

3.1 SNMP Polling

This is a mostly well-understood area, but not quite well enough: we will be doing an *enormous* amount of polling of the local neighborhood, but – curiously – the remote pollers will have very limited knowledge of what they are asking for. This is fully intentional.

Instead, they will poll strictly out of the directory. We'll have what amounts to a list of OIDs for each device, and the poller will query this information and send it back to the mothership. Each bit of data will have a timestamp, the OID, the value or a failure code, and the response time.

Five minutes is the standard polling interval, but it's not a certainty that we'll actually be able to poll everything in that time period. It's dangerous to simply poll until you finish the list, because a more-than-5-min poll means that you run into the next polling period and make *that* run go over.

[8] Me, plus Eric Fleischman
[9] LLNW

This has happened with Cacti before on smithers (especially before we increased the RAM), where an overrunning job causes hours of cascading failures and an enormous load average until an admin can kill everything in the morning. We've lost a lot of data this way.

The pollers will have a hard-stop time at the end of their 5-minute interval, but we recognize that not all OIDs are created equally: each OID requested has the concept of "priority", and the poller will always do the higher priority items before attempting the lower priority ones. This means that if time runs out, we have at least gathered the most important information, leaving the lesser ones for another cycle.

Furthermore, we'll monitor the amount of data we were not able to collect so we can detect nodes that are simply attempting too much and can tune accordingly. But our SNMP poller will be so fast that we won't ever run out of time.

3.2 Netflow data

Collecting netflow data on remote routers involves a *tremendous* stream of information, and we have to take substantial steps to summarize it dramatically before shipping back to the mothership. Taking the full stream of data, we summarize over the same five-minute intervals that we use for standard polling and coalesce the data into

- Timesnap
- Reporting device (router) IP
- Egress interface index
- Octet counter
- AS Path

The octet counters will be rough estimates based on extrapolating from the sampling percentage taken in the router, and on the known lost-sample rate (the config flow capture program knows when samples are missing), but it's not going to be terribly accurate.

But by using the reporting router IP and the egress interface SNMP index, we can tie each interface's samples to the actual bandwidth used on those interfaces, allowing proper attribution of bandwidth to the AS list.

The AS list is simply a string showing the path to the target AS, and the real processing of this summarized data is always done at the main office.

We may increase the period of data collection if we have to, but using the same 5-minute sampling period does simplify reunification with the real bandwidth stats.

3.3 Syslog Data

We expect to have a syslog receiver in each location, and this will all be collected in a holding bin until it can be sent back to the mothership. Very little local processing will be done, and (in particular), we don't do alerting at this level: we merely collect and forward.

3.4 SNMP Traps

As with syslog data, we will collect and forward this data back to the central site for processing, where an application will receive that data.

4 Replicating data back home

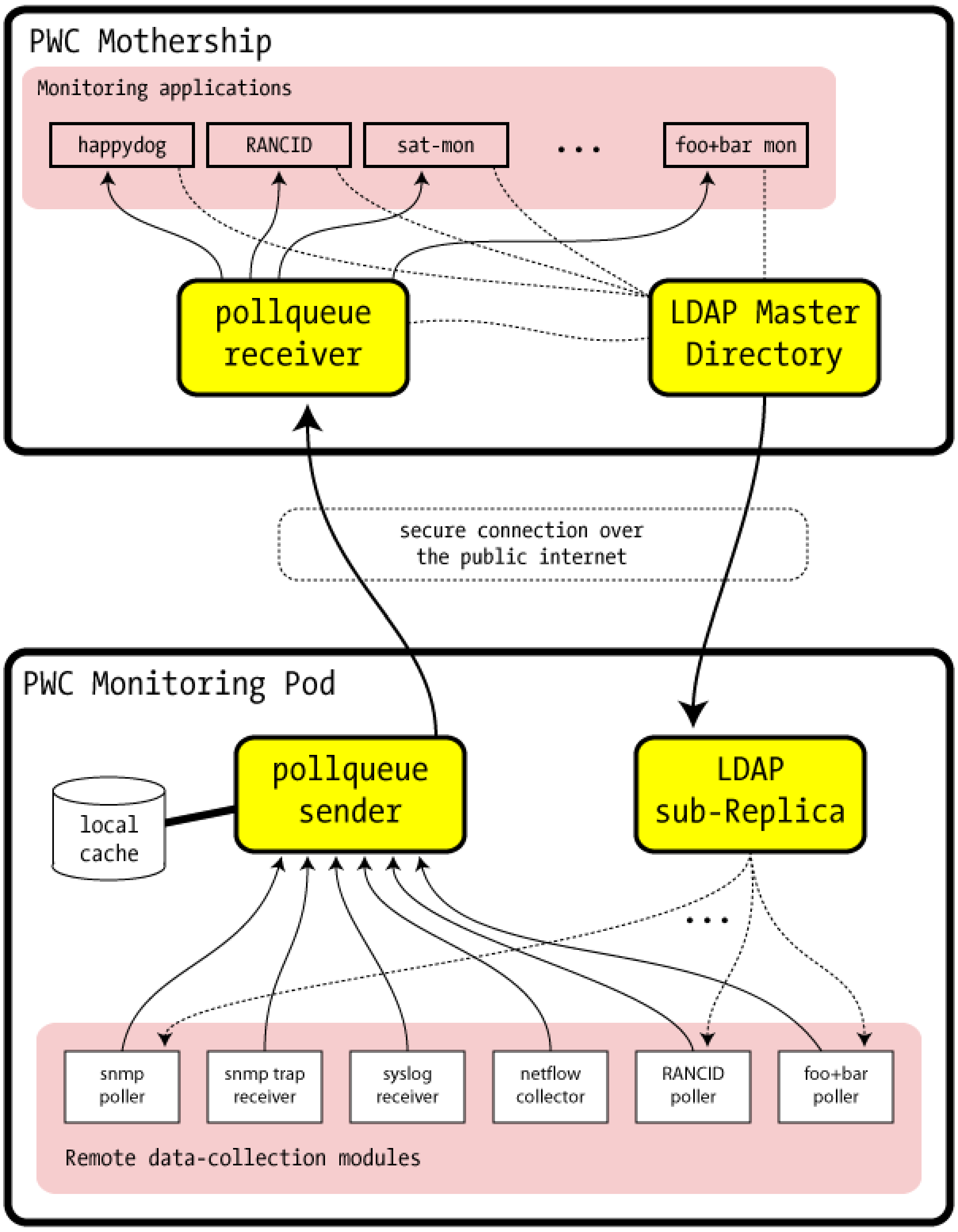
All this data collected remotely has to make it back to the central site for processing, it’s imperative that we create a mechanism with these attributes:

1) Proper queuing in the event of lost connectivity to the central site. We can’t just throw away data if LLNW is down again – the collection pod should keep collecting independent of whether we’re able to talk with the mothership at the moment.
2) A **single**, extensible transport mechanism for replication that *all* polling systems can use. It must be able to pass back SNMP results, SNMP traps, syslog data, RANCID configuration, netflow summaries, and anything else we need to send back.

The obvious – but wrong – way is to have the collector talk directly to its master data store back at the main office, and save the data into a local file if it’s not able to reach it. This is the right answer only if there will only be one application doing the collecting.

Replication is a hard enough problem without having to invent it more than once, and a system (driven by the directory) that allows for queuing of *anything* will be dramatically more flexible. By building a general mechanism, then even new types of data not anticipated can use this same vehicle for transport back home.

The center of this system will be pollqueue data sender and receiver processes connected by a secure link over the public internet: the sender keeps a local cache (perhaps Berkeley DB), and its job is to replicate the data back to the mothership.



The remote monitoring processes *only* talk to the pollqueue sender, and the PWC-hosted applications at the data center *only* get their data from the pollqueue receiver.

5 Central Processing

The mothership is the center of the universe, and it's the master source for the main config repository and all the bulk data in various forms. Most of this revolves around the directory, and part of what is replicated to the pods are directives for polling.

5.1 SNMP Polling and feedback

All polling is based on OID, of course, but it's common for more than one unrelated system to request the same data from a given device. Happydog, sat-monitor, and Cacti all poll more or less the same thing, every 5 minutes, and though this is not the worst of our polling problems, it does not scale well and is wasteful of a precious 300-second window.

Instead, each application wishing to poll "subscribes" to the OIDs it's interested in, and more than one device may well ask for the same OIDs. All that's sent to the pods is a list of OIDs – not the subscribers (the poller doesn't care), and when the results come back, the data is handed off to each waiting application. No application will have any idea that others are asking for the same thing, and once the data is polled at all, it's free to use it for any other application.

This forms a tremendous decoupling of the application from the poller, and it allows the SNMP poller to focus on being exceptionally fast an efficient, not worrying about who's going to consume the data.

The system will support the notion of multiple data gatherers for each device. When we think about the distributed monitoring pods for the Bandcon ring, each location polling only its own data means we could lose a day of data should a machine fail.

Instead, each box will know how to poll not only its own routers, but its neighbors as well (though with a slightly lower priority), and send this all back up to the mothership. The neighbor data is so tagged (as a neighbor), and the central data collector will keep the best data it gets: if we receive the samples from the primary poller, we use it and discard the neighbor polls, but will have fallback data to rely on if we need it.

This will go a long way to never losing crucial billing data.

5.2 Historical archiving

Because of the queueing system that shovels the data back to the main data center as soon as possible, the data-storage needs on the pods will be modest: only enough to hold the polled data while the internet is down.

But because servers always come with much larger drives than needed, we'll have the opportunity to use these collection pods as disaster-recovery archives.

While the replicate-back-home process continues, we'll simultaneously save a copy of the data to a local archive and assume we'll never need it. As the archive reaches some predetermined size or age (say, 20GB or 6 weeks[10]), or when the available hard drive space drops below a certain critical percentage, historical data will be aged out of the archive.

In the event of catastrophic data center failure, we'll at least have a substantial body of data in backup form.

[10] These are completely random numbers – we'll pick what works

5.3 Subscribing to second-order data

Our subscription model suggests that applications can register their interest in particular data from the remote collection pods, but that's not the only information we could be interested. Raw interface counters require a knowledge of history in order to compute actual bandwidth used (the delta from the prior sample), and most applications care about the latter and not so much the former.

Once the interface-counter receivers have converted the counters to bandwidth use, they would publish these stats for further consumption by the rest: saturation alerts, dead-circuit detection, the usual bandwidth graphs – these all consume the processed data.

It's easy to imagine this with most of the counter types.

5.4 Remote dev-query

The current dev-query program makes a realtime SNMP walk of the target device and produces a web page listing each interface's name and characteristics. This has proven to be exceptionally useful for engineers looking for a certain port or find out what's available on a device.

Realtime *remote* dev-query is problematic: we do not really have the ability to proxy from the central server through the monitoring boxes to the target devices, but we still wish we had this.

Instead, the polling process will include (via directory-driven instructions) a request to walk all the interface-specific information from every device at every polling period. It will have a lower priority so that we don't exclude more important information, but we'll have a fresh copy of the interface information on every polling period.

The data is fed back to the mothership via the usual means, and is re-injected back into the directory. Each device in the customer hierarchy maintains a set of "interface" objects (indexed by the SNMP index) which mirrors the parameters on the remote device.

These are all readonly with respect to any user-interface configuration screens – they are populated exclusively by the polling process, not the administrator – but it allows for direct access to highly fresh information that includes everything we can find about these interfaces. Dev-query now becomes a local LDAP lookup[11] rather than an impossible remote SNMP query.

6 Sample Retrofits

We'll touch on three applications which might be retrofitted to use the new scheme just to show the general approach. There are always two general ways to do it: complete application integration, or config wrappers, and we'll touch on both.

6.1 RANCID

This is our config-monitoring system, and out of the box assumes it's running all as part of a single system. Though each "group" – for us, "customer" – has its own subdirectory (`var/myspace/`, `var/Jupiter/`, etc.), the key login information is still centralized in a `$HOME/.cloginrc file`. This specifies the login names, passwords, and other parameters required to connect to the various customer devices.

[11] Each bit of data will be tagged with a refresh time, so the freshness of the data will be apparent

There are numerous problems with a common `.cloginrc`, not the least of which is the completely intermixed sharing of sensitive login information, but in practice it's hard to maintain: making a change to one customer's parameters often has an affect on something else. It's a side effect of the file format used.

Retrofitting RANCID for use with the directory involves several steps:

1) Migrate from a global `$HOME/.cloginrc` to per-customer `cloginrc` files (located in the same per-customer subdirectory as the list of routers to operate on). This was done a couple of weeks ago, and has value even without any migration to a directory-based monitoring system. It involved very few changes to the actual RANCID code itself, because seriously intrusive changes make it difficult to upgrade to new releases down the road.
2) Rather than poll from the LIST_OF_GROUPS variable in etc/rancid.conf, we'll get the list from the directory. This is simply the list of all customers who have any devices tagged to be monitored by RANCID.
3) The per-customer cloginrc file, which is now maintained in the RANCID hierarchy, will actually be move into the LDAP directory and refreshed to the filesystem before each run. LDAP won't really understand the format – it's mainly just lines of text.
4) The `router.db` file, which now contains the list of all devices, their types, and up/down status, will be likewise refreshed from the directory at each run.

The normal RANCID run out of cron every six hours will simply have a small pre-processing step to prepare the hierarchy for a standard "`rancid-run`" command, and the tool itself will have very little knowledge of LDAP. It will simply find its data where it expects, and will have no idea that humans aren't editing the files by hand.

Note: these steps are to LDAP-ify the RANCID process, which is not the same as preparing it for remote polling. That's going to take a bit more work (we have to decouple the config polling mechanism from that which does the diffs and SubVersion checkin), but the directory is likely to make this easier than working with the stock RANCID code.

6.2 DNS

DNS for the `*.cust` zones is now maintained on `mrburns` in the same place with all the other domains we're authoritative for: in `/chroot/named/conf`. These zones are maintained by hand and updated as devices come and go.

But since the directory will contain this information anyway, it's foolish and dangerous to perform double entry. Though it's possible to build a version of BIND that obtains its zone data from LDAP – and this may be worth doing someday – we're instead going to take a simple wrapper approach.

A script on the nameserver will extract the DNS zone data from the directory, create the zone files, and kick `named` when required. This allows for a cron job to keep DNS refreshed from the directory at regular intervals, or updated on demand.

No changes to DNS software itself will be required, though PWH's DNS tools (especially the use of `Masterfile`) will require some tuning.

6.3 Happydog

(fill this in – but just for illustration, not necessarily proposing to actually do it)

7 LDAP Schema Design

This is the hardest part of the project: the directory schema defines the operation of the entire system, and there is much we don't know about how it all fits together.

(more more more)

8 Obvious questions

This is presenting an entirely new paradigm for building a monitoring system, and it's likely that questions come to mind on how to make it all work.

8.1 Doesn't this mean that everybody has to learn LDAP?

Not any more than users have to learn SQL to use our existing applications. There may be a bit of notational learning for those doing the in-depth maintenance (say, adding new devices or moving things around), but there's no reason that applications can't hide that from the users.

The *developers* will certainly have to dig deeper into LDAP, but we'll have plenty of examples of how it works, it's not really that different from SQL, and it will be useful for them to get their heads out of the relational-only world anyway.

8.2 Doesn't this mean each application has to talk to both MySQL and LDAP?

Yes, but they'll mostly have to do the equivalent anyway. Unless the entire PWC monitoring suite is built into a single monolithic database, there will already be multiple persistent stores to talk to. The principle of customer partitioning demands some kind of separation between one customer and another, and this requires (at minimum) separate tables, and probably separate databases.

If we have to talk to multiple SQL databases – which already breaks db-enforced referential integrity – then it's no more difficult to talk with one SQL database and the LDAP configuration store than it is to talk with a pair of SQL databases.

8.3 This doesn't support direct access to remote devices, does it?

No; this is a different problem to solve, and is not necessary for this project anyway.

Our goal is to perform robust, unbreakable data collection, and since we realized very early that we cannot do this remotely, we didn't solve the problem of remote, direct access.

This means that useful programs like `dev-query` and `dev-survey` can no longer be supported, but we expect that we can simulate this with collector programs that query the various OIDs and replicate them to the master directory. Then dev-query becomes a directory-query program rather than an SNMP query program.

PWC will use our data collection systems as jump boxes, so we expect to provide easy access to the engineers.

Furthermore, it's not out of the question to build some kind of remote VPN or proxy system, but this is a harder problem than it looks. It's likely that we can live without this, or solve it as a separate problem from the monitoring system.

8.4 Aren't some applications going to be harder to build or retrofit?

Yes, of course.

It's likely that *new* applications will be designed and built with this paradigm in mind (and will leverage a growing common library), but older applications could be much more challenging. We expect, for instance, RANCID to be relatively straightforward to retrofuit, but Cacti to be much more difficult.

We're solving a hard problem, and it's not surprising that distributed data collection won't fit with some of these applications. We may find that it's better for us to simply rewrite an application like this than to try and retrofit (and continue to retrofit) this application.

8.5 Will MySQL be on the monitoring pods at all?

I hope not. MySQL is high maintenance and high footprint, and it's not clear that it will provide anything useful on these pods. We plan on relying on the LDAP directory to drive the configuration, plus a lightweight data store to hold the collected polling data, and that's probably Berekely DB. It's designed for the kind of sequential access we require.

There won't ever be the need for general SQL-type queries of the collected data, only specific queries for which the indexes will be optimized. MySQL wouldn't provide anything we require.

However, MySQL will be heavily used by the applications back at the data center, because we *will* require that kind of *ad hoc* query access. It's exactly the right answer for this purpose.

8.6 Will *.cust DNS be used on the monitoring pods?

The monitoring system won't use it, though PWC engineers may want it when using the machine as a jump box.

The LDAP directory already contains everything about all hosts we monitor, including their IP addresses, and it seems pointless to introduce another point of failure to provide a name-to-IP mapping we already have.

It also might introduce some timing issues which would be hard to track down: if an IP address is changed in the directory, this should be *authoritative* for everything about the device, but if DNS and LDAP don't replicate at the same time, it could mean some really odd results.

It seems really unfamiliar to think this way, but I can't find an upside to using DNS at all.

Note: this is *only* about how the monitoring applications query the directory; all the user-visible interactions will use readable hostnames.

8.7 Could we use this system for "regular" centralized authentication?

I hope so. This is the traditional use for LDAP – a central user directory – so that all logins to all systems (happydog, Cacti, IPPlan, our tools, jump boxes, etc.) operate out of a single directory so that – for instance – terminating a staffer can be done in one place instead of two dozen.

We’ll probably start by integrating LDAP into HTTP authentication that now protects our happydog URLs: this will allow us to define the customer credentials in one place (the directory), and slowly expand to other places.

I’d imagine that the *last* place we’ll get to using LDAP auth is the Linux/BSD boxes for ssh logins (though it certainly is appealing).

8.8 Can LDAP be backed up?

Absolutely. Not only can one simply copy the underlying database files to some other system, but it’s straightforward to query the directory remotely via the standard tools and get a dump of everything. There is a standard (RFC2849) output format known as LDIF[12], and it’s all readable ASCII. One user’s information:

```
# Jeffrey Papen, SBSUsers, Users, MyBusiness, peakweb.lan
dn: CN=Jeffrey Papen,OU=SBSUsers,OU=Users,OU=MyBusiness,DC=peakweb,DC=lan
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: user
cn: Jeffrey Papen
sn: Papen
description: Mobile User
givenName: Jeffrey
distinguishedName: CN=Jeffrey Papen,OU=SBSUsers,OU=Users,OU=MyBusiness,DC=peak
 web,DC=lan
displayName: Jeffrey Papen
homeMTA: CN=Microsoft MTA,CN=MAYORWEST,CN=Servers,CN=first administrative grou
 p,CN=Administrative Groups,CN=PEAKWEB,CN=Microsoft Exchange,CN=Services,CN=Co
 nfiguration,DC=peakweb,DC=lan
proxyAddresses: smtp:jeff@peakwebhosting.com
proxyAddresses: smtp:jeff@peakwebconsulting.com
proxyAddresses: smtp:jeff@papen.com
proxyAddresses: smtp:jpapen@peakwebconsulting.com
proxyAddresses: smtp:jpapen@peakwebhosting.com
proxyAddresses: smtp:jpapen@papen.com
proxyAddresses: smtp:jeffrey@peakwebhosting.com
proxyAddresses: smtp:jeffrey@papen.com
proxyAddresses: X400:c=US;a= ;p=PEAKWEB;o=Exchange;s=Papen;g=Jeffrey;
proxyAddresses: SMTP:jeffrey@peakwebconsulting.com
homeMDB: CN=Mailbox Store (MAYORWEST),CN=First Storage Group,CN=InformationSto
 re,CN=MAYORWEST,CN=Servers,CN=first administrative group,CN=Administrative Gr
 oups,CN=PEAKWEB,CN=Microsoft Exchange,CN=Services,CN=Configuration,DC=peakweb
 ,DC=lan
mDBUseDefaults: TRUE
mailNickname: jeffrey
name: Jeffrey Papen
objectGUID:: LpwDTH0m1kW46YAonOXPiw==
codePage: 0
countryCode: 0
primaryGroupID: 513
objectSid:: AQUAAAAAAAUVAAAAf7J25lYErbU/PNNOiQQAAA==
sAMAccountName: Jeffrey
```

All objects are dumped in this similar way, can be reloaded into this directory via this format, and it’s like a documented `mysqldump.` It’s really easy to automate too.

[12] Lightweight Directory Interchange Format; http://en.wikipedia.org/wiki/LDIF