When you think of DHCP, where do your thoughts tend? Perhaps to assigning temporary addresses to devices. Perhaps to lease maintenance and netblock assignment. Perhaps as the original extension to BOOTP [1]. DHCP is a lot of things to a lot of people, sometimes maligned, often underappreciated, and quite often used without full comprehension. After an introduction, I plan to point out some interesting possibilities, implementation issues, and potential novel uses for DHCP that most sites likely haven't considered. At the end, I’ll address our implementation and how it increases our operational efficiency.

Introduction
The Dynamic Host Configuration Protocol (DHCP) [2] was defined in October of 1993 as an extension to the Bootstrap Protocol (BOOTP) then prevalent. Since those primordial days, it has undergone many revisions, extensions, and has become de rigueur at most sites. The original raison d’être of DHCP was to enable one to dynamically assign addresses to a pool of machines. This was a boon for ISPs and corporations that had mobile machines or a limited set of modems where users would dial in, set up a PPP or SL/IP connection, and be assigned an IP address. Enter DHCP, a way to give a lease on an address to a connection that could be reused by somebody else later. But DHCP has many more extensions over BOOTP as well. I’ll add some color to these shortly.

Later, in 1999, DHCP got a new lease on life (so to speak) with the invention of Preboot eXecution Environment (PXE) [3] by Intel and SystemSoft. PXE was, and remains, a marvelous invention allowing for embedding DHCP (and TFTP, or Trivial File Transfer Protocol) into the NIC so that it can download some small bit of code to enable bootstrapping and installing machines. The combination of DHCP with TFTP [4], and some identifiers like UUID and GUID with an API, was a master stroke for the sites that needed a way to install machines with as little interaction as possible. To this day, TFTP is still used even though it was originally implemented in 1981! These days, though, TFTP is often used as a means to load a small executable like PXELINUX [5], which then will do some further loading via HTTP or NFS or another mechanism. Even so, TFTP still has widespread use in the network provider space. But this article isn’t about either PXE or TFTP.

The Workings of DHCP
DHCP has a small number of things that are mandatory to send to the host and come right at the top of the request and reply packets. Many are carried forward from BOOTP to retain backward compatibility, with some minor changes to extend the capabilities. The important ones are outlined in Table 1.

Let’s briefly refocus on the “Options.” Many of these are defined in their own RFCs. Some are vendor-specific tags that extend the protocol in particular ways, like defining specific addresses of servers for particular protocols. Some are functionally obsolete, like “NDS Server” or “Impress Server.” I will have more to say about options later.

It’s important to note that DHCP protocol has three modes for assigning addresses: automatic allocation, dynamic allocation, and manual allocation. Automatic allocation assigns a
permanent IP address to a client from an address pool. Dynamic allocation assigns an address to a client from a pool for a limited period of time, which may be renewed. Manual allocation allows a network or system administrator to map the MAC address to a specific IP address. Dynamic allocation is the mode that allows automatic reuse of IP addresses for transient clients.

On DHCP Options

Every DHCP option consists of two bytes. The first byte is the option identifier, or tag. The second byte is the length of the option data in bytes. So you can see that there are built-in limits on options. There can be only 256 options and every option can be only up to 255 bytes long. A length of 0 is valid for Boolean type flags, and option tags 0 and 255 are special. Thus, since an IPv4 address is four bytes, every option tag that references an IPv4 address (TFTP server, DNS server, etc.) is four bytes long (plus the two preceding bytes with the tag number and the data length).

Other interesting tidbits:

- Some string options are required to be NULL terminated. Others are not. (The null is included in the length!)
- As mentioned, Option 255 is special and has a 0 byte length. It is added by the server to signal the end of the DHCP reply.
- Option 0 is also special and is a pad as defined in RFC 2132. It is used to cause subsequent fields to align on word boundaries if necessary.

- Option 55 carries a parameter request list. It is inserted by the client and includes the tag numbers of options that the client wishes to receive. The byte length field of Option 55 holds the count of the number of tags, and each byte after the length is an option tag number. The DHCP server isn’t required to answer every tag if it doesn’t have information for that tag, but it must try to provide any tags that it does answer in the order requested by the client. Thus, if the client requests options 5, 120, 40, 35, 39, 20, then the server, when composing its reply, should insert them in the reply packet in that same order.
- Wireshark will decode options for you. It is quite illuminating.
- Option 82 is quite interesting, and figures heavily into our implementation. Stay tuned.

What Do We Need?

With the obligatory introductory technical information out of the way, let’s take a step back to look at business objectives. At my organization, we have a number of custom supercomputer machines for molecular chemistry research. Each machine is functionally identical, with a number of ASIC boards, a number of commodity computers connected to the boards with a custom PCI card, and a number of commodity network switches connecting the off-the-shelf computers into the network for storage and supporting software.

When a commodity node fails, we want to replace it quickly. The new node must have the same name, same location, and same IP address as the old one, because it will be doing the same thing. Also, if a switch fails, we want to be able to put a new one in place, plug it in, and have it auto-configure itself, including the correct VLANs, switch port labels, and sundries. It is impractical to have a spare sitting around preconfigured for every possibility of failed device. It is very desirable to have a factory-default box ready to plug in, install itself, and be ready to go in a couple of minutes. DHCP can help manage this. It was on the search to solve these issues that we discovered Option 82.

Additionally, my organization tends to buy servers in units of a rack, to save time, labor, and money on partial integration. This allows us to take advantage of third-party integration services for getting all of the machines cabled, labeled, IPMI addresses set, tested, and ready to deliver; integrators pre-stress all manufacturer-delivered machines for early failures. Once all of the preliminary work is done in this fashion, we can receive the entire bundle in a crated rack, roll it into place, plug in the power distribution unit and the network uplink cables, and install all of the servers at one time. Because all of the IPMI addresses are set, all we need to do is configure the switch(es) to the appropriate VLAN(s), connect the power and network uplinks, and start installing all of the machines remotely.
Musings and Hacks on DHCP

One last component of labor and delivery speed that caused us to expend a considerable amount of time and effort was the fact that, to install the nodes, we needed to assign IP addresses. Referring back to the DHCP “modes,” DHCP Dynamic mode is unfavorable because we want all of the nodes to have constant DNS resolution. We could use dynamic DNS for this, but adding that infrastructure and having IPs and names in a predefined order to facilitate subsequent debugging has positive benefits and fewer moving parts. Any statistics or history that is indexed by an IP address when using dynamic mode would be lost on a random DHCP reassignment later on.

Up to this point, we’ve had our integrator supply us with a list of all of the MAC addresses of every node in a rack integration spreadsheet. This spreadsheet has the MAC address of each eth0, eth1, and IPMI for each server, its rack position, name, serial number (for later RMA), Ethernet switch port designation, switch name (if more than 40 nodes in a rack), and PDU receptacle (if switched PDU). The MAC address collection in particular adds a lot to delivery time because it requires the integrator to gather all of the extra MAC data and carefully collate it; serial numbers are generally more accessible and do not require booting the machine. It also means we need to add all of the MAC addresses into isc-dhcpd. In our case, that means populating them into a database with the host record, so it’s not that hard, but it is tedious and time-consuming and leads to slower delivery and installation.

The most common DHCP server in use today is the Internet Systems Consortium’s dhcpd [7] (isc-dhcpd). It has a number of interesting features: for instance, client groups and subgroups, support for dynamic DNS updates, and access lists. Until now, we have been vigorous users of isc-dhcpd for our commodity node installations. Then our eyes were opened with the previously undiscovered utility of...

Option 82

DHCP Option 82 is defined in RFC 3046 and is a bit of a lesser known gem. It was devised, in part, as a solution for cable modem, DSL, and other providers with a high port count that want to assign IP addresses to a particular subscriber line statically without having to worry about exceeding dynamic capacity of a pool. It is still heavily in use today by that same contingent, but with some growing use outside of that. The rest of this article discusses our attempts to use and our final application of Option 82. But, first, I’ll go over a few technical details.

Option 82 is called the Relay Agent Information Option. A relay agent is any device that listens for DHCP requests on one subnet and forwards them to a DHCP server on a remote network. DHCP is inherently a Layer 2 protocol and uses link-level broadcast technology to find a server. Relay agents enable DHCP to be Layer 3 capable. Relay agents are in common use at sites where there are many VLANs (or subnets) because it prevents the need for having a DHCP server per VLAN. Relay agents (typically managed network routers) are, among other things, required to add a gateway address into the request so that the server can send the reply back via the relay agent.

One uncommon thing about Option 82 is that it includes two sub-options. The length byte of Option 82 contains the combined length of the two sub-options and their sub-tags. Because there are currently only two sub-options (they automatically have room for up to 256), these are tagged 1 and 2. Sub-option 1 is the Agent Circuit ID and sub-option 2 is the Remote ID. Each of the two sub-options is voluntary. A relay agent may insert one or both into a DHCP request or skip it entirely. Additionally, the encoding of a sub-option is left up to the implementer. A DHCP server must include the entire option in its reply, verbatim.

◆ Agent Circuit ID: This is intended to be used as a port identifier for the agent upon which the request was received. It is left up to the vendor to determine how to encode the port (e.g., ASCII or integer, numeric, or alpha-numeric). Some vendors include VLAN and blade information for chassis switches along with the port, while others include only a port name or number. Cable modems often include the virtual circuit of the subscriber.

◆ Remote ID: If included, this signifies the device acting as the agent. Again, it is not specified in the RFC how this should be encoded, so it could be an ASCII switch name, a hexadecimal encoding of a serial number or VLAN IP address where the request is received, or a caller-ID for a dial-up connection. Because the RFC requires the remote ID to be globally unique, many vendors use an encoded or literal serial number but allow the administrator to override this with an arbitrary name. We use the unique name of the switch where possible.

In big networks, it is not uncommon to have multiple levels of relay agent between the source network and a destination server. Different switch vendors handle this in different ways. Some allow you to append or overwrite the remote ID and circuit ID information with new information, at the administrator’s choice. Some will only overwrite. Check your switch documentation to learn more.

You may be wondering to yourself, okay, he’s explained what all of this agent stuff is, but what does it give me that I didn’t already have? I’m not a cable modem provider, what’s in it for me? Excellent question. Let’s relate it back to my use case.

Option 82 allows me to say the request that arrived tagged with switch name rack201-sw on port 32 will be given IP address 10.10.1.9. Or, the device that requested an IP and bootfile on port 16 of mgmtswitch1 is going to be rack216-sw with IP 10.10.1.1 and should bootstrap its configuration and self-configure onto
The host-identifier option can be used to specify "use this
Sub-classes present another way to group attributes:
Classes are a way to group attributes:
dhcpd configuration and its limitations:

- Classes are a way to group attributes:
  - Classes are implemented as a single linked list data
    structure.
  - Classes can be used in the host block to decide whether to
    assign an address.
  - Classes cannot be nested.
- Sub-classes present another way to group attributes:
  - Sub-classes are stored efficiently in a hash.
  - Sub-classes can be instantiated dynamically at runtime
    as an arbitrary group of attributes pulled from the DHCP
    request (class followed by sub-class).
  - Sub-classes cannot be used in a host attribute to assign an
    address.
  - Some people have devised source code patches to make
    sub-classes useful in the host stanza, but they are against
    old source code and not integrated into the core.
- The host-identifier option can be used to specify "use this
  other thing as a MAC equivalent." This would appear to be a
  way to get there, however:
  - You can concatenate various objects dynamically in an
    isc-dhcpd stanza, like remote ID and circuit ID, but objects
    created this way cannot be used as host identifiers. They
    are useful for logging, but not allocation.
  - You can specify that the host-identifier is either a re-
    mote device ID or a remote circuit ID, but not both nor in
    combination.
    - Remote ID is not useful by itself because it’s just the
      relay agent device forwarding the request. You can’t
      make any useful determination based upon that, unless
      it has only one device plugged into it (you can see why
      this could be okay for DSL and cable modems).
    - Circuit ID is useful as long as you only have one switch
      in your entire network acting as a relay agent.
    - To be effective, the IP must be a combination of the two.
  - You can use an isc-dhcpd hack that says “give this request-
    ing device a dynamic IP address in this particular range,”
    where the range is something like 192.168.1.2–192.168.1.2.
    However, without a way to combine this with circuit ID
    and remote ID, it has limited use.

Here is my summary about why I find this so disappointing:

We could possibly use classes, if we could concatenate the
Remote ID and Circuit ID together to make a unique client design-
ator, but classes are inefficient at that scale, and you cannot use
concatenated objects in such a way.

We could possibly use the host identifier, if isc-dhcpd let you use
concatenated/generated objects.

We could possibly use the dynamic sub-class spawning facility
(remote ID = class, remote-circuit = subclass), but you cannot
use sub-classes in host stanzas.

Our Solution

Frustrated and stymied, we went on an isc-dhcpd-alternatives
discovery trek. We found a basic Ruby DHCP implementation
and extended it to fully support the notion that IP addresses
were to be given based upon the remote ID and circuit ID pre-
sented by the relay agent. Also, because we have all of the infor-
mation about all of our hosts in a MySQL database, we extended
the relevant network table to include remote ID and circuit ID
fields. The Ruby daemon contacts the database upon receiving
a query for a given remote-circuit combination, and then serves
the host an IP address, network accoutrements, and bootfile,
if necessary. This allows us to do complete zero-configuration
installation of a switch for the impending supercomputer rollout
in 2014, about the time this article will be published.

Furthermore, this implementation has led us down the natu-
ral path of considering this for racks of servers. The integrator
can save many hours of work collecting and collating MAC
addresses because we no longer have to care. We can plug in
the switch, let it configure itself, then turn on all of the hosts.
They will all get a name and address according to their posi-
tion on the switch, which dictates their position in the rack. The
DNS is preconfigured. Just before publication we deployed two
racks of 144 machines each using this DHCP server to assign
addresses based upon switch and port. It was a happily success-
ful proof-of-concept.

One last feature of the server is that we can log the MAC address
heard in the DHCP request into a MySQL table associated with
the network entry for the host. This makes it easy to determine
if a given host has moved from one switch to another, or where a
particular MAC address lives if we happen to lose track of it. It’s
a built-in history mechanism and will allow us to track a device
that may have been returned for repair to the vendor and then put back into service as a different node later.

The DHCP server configuration file is very simple. It consists of things like the database table, host, username, password, what logging level to use, the interface to bind to, and any site- or network-specific overrides for testing. All of the logic about IP addresses, ports, subnet masks, hostnames, etc. lives in the database, where it belongs. There is essentially nothing to distribute for multiple redundant servers.

Summary
We can have redundant MySQL databases and redundant servers without the need to worry about generating or keeping DHCP configuration files up to date. We can leverage our inventory database (along with minor extension) with our DHCP address allocation and also keep track of the motion of assets that have been repurposed or repaired. Although I have not yet published the source code for this server anywhere, I can possibly share the code upon request. Some things in it are still a bit site specific (like the database table schema), and it so far has only been tested against Dell/Force10 and HP Procurve switches as relay agents. Some documentation cleanup and further testing is in progress.

Lastly, DHCP options are quite powerful. You may find an interesting gem in there that you didn’t expect if you take a moment to review them. Option 82 is useful whatever DHCP server you use.

Epilogue
Just before sending this article for final publication, ISC released a new alpha version of isc-dhcpd that reportedly allows one to make an Option 82 determination in a class that includes both the remote ID and circuit ID, something like this:

```
class "1-2-3-9" {
  match if option agent.circuit-id = "1.21.1.4/Ethernet9";
}
```

The remote ID and port get separated by a / . Unfortunately, we were not able to test whether this works. If so, this would reduce the major shortcoming of needing something that can actually deal with both remote ID and circuit ID at the same time and would just leave the burden of generating and distributing a very large configuration file with every host defined in it, and the inherent scalability concern of a single large linked list with every possible combination of switch and port.

References
  /bootp-dhcp-parameters.xhtml.