Experience Implementing an IP Address Closure

Ning Wu and Alva Couch – Computer Science Department, Tufts University

ABSTRACT

Most autonomic systems require large amounts of human labor and configuration before they become autonomous. We study the management problem for autonomic systems, and consider the actions needed before a system becomes self-managing, as well as the tasks a system administrator must still perform to keep so-called "self-managing systems" operating properly. To understand the problem, we implemented a prototype self-managing "IP address closure" that implements integrated DNS and DHCP. We conclude that the system administrator is far from obsolete, but that the administrator of the future will have a different skill set than those of the present, focused around effective interaction with closures rather than management of individual machines.

Introduction

Imagine that you are asked to set up a new DHCP/DNS infrastructure. You proceed to collect a pocket-full of "Ethernet keys" that look like USB keys, but each contains a micro-controller and an Ethernet interface, where power is drawn from the Ethernet plug. You proceed to plug one of these keys into a test network and give it a specification of your network architecture in the form of an operating policy. Then you plug in the other keys to the same network, and each copies the policy from the first key. Finally, you unplug some of the keys and plug one or more keys into each Ethernet subnet and *voila*, you have a self-managing IP address infrastructure that is self-healing, and in which telling any key about a policy change causes that change to propagate to the whole infrastructure of keys. If a key dies, you replace it with another key that has – for awhile – been plugged into any subnet containing a working key. No backups are necessary; the infrastructure is completely self-managing and self-healing.

Are we dreaming? Not really, as this paper will show. It is possible to implement such devices and infrastructure. But there is a larger question that we have not addressed: what happens when something goes wrong? The subtlety in managing such an infrastructure lies in the interface between the keys and the human administrator. When things go wrong, a human is still required to intervene and repair problems.

Closures

Our implementation of the above-mentioned behavior is based upon the theory of closures. A *closure* [3] is a self-managing component of an infrastructure that protects one part of IT infrastructure while making its needs known to other closures [20]. The closure model of a managed system expresses the system as a composition of communicating components. In previous experiments on closures [20], it has been demonstrated that any high-level closure needs support from other low-level closures. For example, consider a web service. The service itself can be encapsulated within a closure, but does not handle IP address assignment and DNS [15, 16]. These functions must be handled via one or more lower-level closures in a self-managing infrastructure.

In this paper, we describe experience and lessons learned in building and testing an "IP address closure." This closure is a self-managing "fabric" of distributed nodes that handles DNS and DHCP for an infrastructure. The IP address closure handles address assignment based upon three inputs: requests for addresses, a policy on address assignment, and architectural information about routing and gateways within the network. The IP address closure sits between the web service closure and a routing closure (which may be implemented by a human being), accepting inputs from both (Figure 1).

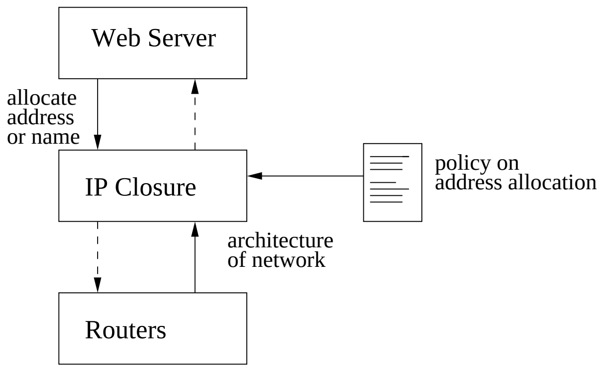

Figure 1: Interaction between closures.

A New Management Style

Managing the IP address closure is very different than managing the Web service closure. The web service closure is managed via "commands" that change state of a single server or web farm. By contrast, the IP address closure fabric is composed of small, movable "black boxes" that can serve as DHCP and/or DNS servers. These are configured by a process of *seeding*. Each box is initialized by physically plugging it into the same subnet as an already seeded box, by moving it physically. The new box discovers the existing seeded box, clones its configuration, and receives an idea of network topology, policy, and locations of peers from the existing box. After this, it is moved to its final physical location, after which it can serve to seed other boxes.

Simple Hardware Components

An element of the IP address closure is an extremely simple device with an Ethernet connection and some form of persistent storage. It is conceivable that a closure node could be implemented in hardware using only non-moving parts such as flash and regular memory (no hard disk would be required), thus leading to extremely low hardware cost for the self-managing fabric. It is even possible to power a node from the Ethernet connection, so that it can be a completely self-contained device similar to a USB key (an "Ethernet key"). We foresee a time in which IP management could literally be accomplished by a pocket full of keyring-sized devices, carried from room to room as needed. A similar approach, using the same environmental discovery and arbitration algorithms, could be used to create closures for other tasks such as distributed monitoring, intrusion detection, troubleshooting, web caching, file system caching, and secure remote access.

Backup and Recovery

This is a dramatic difference in how one provides failover and recovery in the IP address closure fabric, compared to managing current DNS and DHCP servers. To establish redundancy on a subnet, one simply plugs another box into the subnet, and the new box makes itself a clone of the boxes it discovers, to become a backup server. If one unplugs a box, any backup servers automatically start serving requests. If a box fails, one simply unplugs it and plugs in another. The boxes serve as their own backups; any box is interchangeable with any other in case of failures. Each box discovers where it is operating, and how many neighbors it has, before deciding to provide services or serve as a backup server. Thus backups are as easy as keeping a spare box that one plugs into a subnet periodically in order to keep the backup node up to date, and recovery is a matter of plugging the backup node back into the network so that its changes can be propagated to other nodes.

Low-level Last

It may seem to the reader that we have gone about the problem of building closures "backwards"; previous authors have studied "high-level" closures that – to operate properly – require low-level closures that we tend to implement *after* the closures that utilize them. The reason for this backward implementation order is that many of the challenges in building a closure come into play at the lowest levels, where the interface between the system administrator and the closure is most complex. At the lowest level, closures are limited by the fact that software cannot accomplish physical changes in hardware or network configuration. When configuring a web server [20], this is not much of a concern, but at the IP level, it is a central issue.

Related Systems

In large-scale systems, manual procedures for maintaining static and dynamic IP address assignment are both tedious and error-prone. IP management tools have been developed to help administrators manage the IP space in an enterprise; CISCO Network Registrar [2], INS IPControl [6], and Lucent VitalQIP [13] are examples of current products. Common features of IP management software include integrated DHCP and DNS service, centralized policy management, and failover mechanisms for high availability. These products require crafting of detailed IP assignment policies, as well as manual configuration of all nodes included in the service. Melcher and Mitchell [14] mention the need for an autonomic solution for DHCP, DNS, LDAP, and other services. It is also highly desirable to minimize the amount of human input necessary to configure the system, avoiding the "incidental complexity" of making policy decisions that have no externally observable behavioral consequences [3].

Goals

Our goals in creating the IP address closure were to help administrators by:

- Encapsulating a reusable design of the IP assignment plan in a policy.
- Reducing incidental complexity by automating unimportant decision-making.
- Automating the process of implementing changes in policy.
- Providing autonomic features such as self-configuration, self-backup, and self-healing.
- Simplifying day-to-day management of the IP address (DHCP/DNS) infrastructure.

The IP address closure can be seen as an effort to implement autonomic features [4, 5, 10] in the IP layer.

Paper Organization

In this paper, we will use the IP address closure as an example of the potential impact of autonomic systems upon system administrators, and show that system administrators can benefit from it and similar systems. Far from threatening the jobs of system administrators, the IP address closure is instead a "partner" that requires ongoing management, in return for offloading some common management tasks.

This paper is organized as follows. We begin by describing the overall design and function of the IP address closure. We then discuss the design and implementation details for the IP address closure and critique our prototype. We subsequently discuss the relationship between autonomic systems and system administration and then discuss the issue of exception handling. Finally, we conclude this paper and discuss future work.

Closure Design

The design of our IP address closure is so unlike that of any prior work that a detailed discussion of its theory of operation is necessary. In this section, we give a detailed theory of operation intended to convince the reader that the closure will work as described. The closure's theory of operation is somewhat subtle, and this section can be skipped without loss of continuity if the reader is not interested in implementation details.

Peer-Peer Architecture

Unlike prior closures, which resided primarily on one machine, the IP address closure resides within a peer-peer ''fabric'' of distributed ''black boxes'' that manage the state of the IP layer for an enterprise. These ''Peered IP'' management nodes, or ''PIPs,'' manage themselves based upon a high-level policy and environmental factors that the PIPs discover through direct probing of their environments. PIPs can be implemented as small and cheap ''Ethernet appliances'' that support each other and implement both self-healing and self-replication features.

A peer-to-peer solution is more robust and easier to use; there is no need to manage a centralized database. The distributed nodes have a better view of the environment than a central probe; they can see through firewalls and other protections, and can acquire environmental information [12] that is more accurate than relying upon human input. If we tell one peer about a new policy, it distributes the policy to all of its known peers, which continue relaying the policy until it is present on all nodes. However, control of information distribution is more difficult than in the centralized case. For example, at any particular point in time, there can be conflicts between the policy information in replicas. A policy change must be broadcast to all the PIPs.

It would have been nice if we could have utilized an existing peer-peer scheme for implementing our closure. The drawback of utilizing existing peer-peer schemes is that their own bootstrap protocols require prior existence of a stable IP layer. Also, their complexity and goal of distributing large amounts of information is much more ambitious than we need. We utilize a simple pull-only gossiping protocol to communicate relatively brief policy information among PIPs, after a (rather complex) bootstrap and environment discovery protocol that is necessary because there is no IP layer before the closure becomes functional.

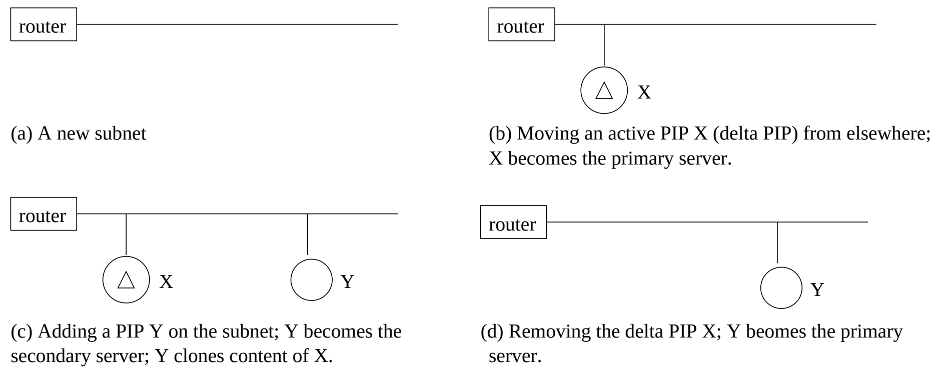
router
router
X
(a) A new subnet
(b) Moving an active PIP X (delta PIP) from elsewhere; X becomes the primary server.
router
X
Y
router
Y
(c) Adding a PIP Y on the subnet; Y becomes the secondary server; Y clones content of X.
(d) Removing the delta PIP X; Y beomes the primary server.

Figure 2: Bootstrapping a new subnet from a delta PIP.

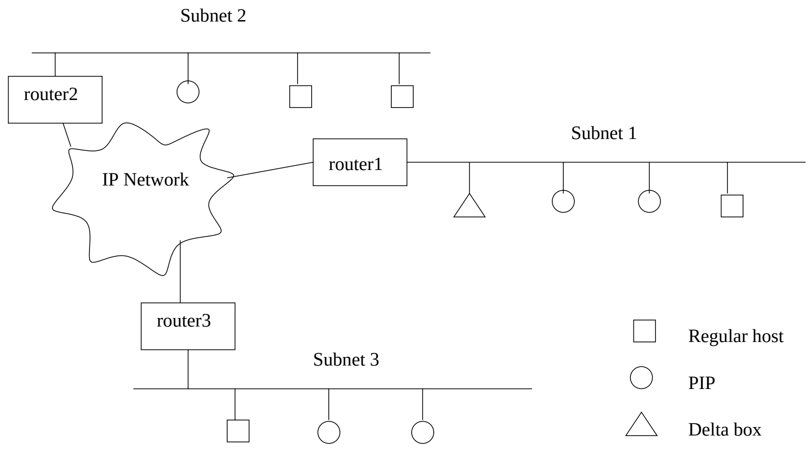

Figure 3: An example of how PIPs can be deployed.

In practice, using a complete peer-to-peer environment poses problems in network design. If a network policy chooses to maintain separate subnets, it may still allow the DHCP servers to talk to a central server, and vice versa. If administrators choose to use a peer-to-peer architecture, the complexity of firewall rules will be increased. The deployment of web services [23] faces similar issues. If lower-level network closures exist, the requirement of configuring firewalls can be delegated to it; if not, administrators must manually configure the firewalls.

Bootstrapping the Closure

The most innovative feature of our IP address closure is how it bootstraps itself into a functional state on a previously unmanaged network. Unlike other closures and autonomic computing solutions, our closure must be able to cope with a network where the IP layer is not yet functional. This leads to a rather unique process for bootstrapping, based upon policy seeding and environmental discovery. There are three types of hosts in an IP address closure: regular host, PIP, and delta box(Figure 3). Regular hosts are the clients of the DHCP service provided by PIPs. PIPs are the management nodes that provide DHCP and DNS services. A delta box is a special type of PIP that potentially contains different information from other PIPs on the same subnet; otherwise, it is the same as a generic PIP. A delta box can be used to deliver information to another subnet by connecting it physically to that subnet. This feature is very useful for distributing policies to networks that are physically segregated from the rest of the infrastructure, e.g., by firewalls.

The bootstrapping process for the IP address closure is different from that for a normal system. A PIP, referred to as "delta PIP," can be moved *physically* through the network to propagate information about changes. Bootstrapping of the closure is done by a combination of logical specifications and physical moves of devices. To bootstrap the closure, one starts with a policy file and loads it into a single delta PIP. Connecting this box to each segment of the network discovers other peers on that segment and communicates the policy to them. The delta box also records and distributes the addresses of other peers. At the end of the seeding process, every node knows about every peer, and the delta box is no longer needed. The bootstrapping process is depicted in Figure 2.

Figure 3 shows an example of the deployment of PIPs. The IP network could contain many firewalls between routers, and the subnets can even be disconnected with no ability to communicate, provided that physical moves are utilized to propagate policy changes. Assume subnet one is separated from subnet two and subnet three. Subnet one has three PIPs deployed (two PIPs and one newly arrived delta box). Subnet two has only one PIP, so there is no failover backup on subnet two. Subnet three has two PIPs that will form a failover pair.

The one command that every PIP must understand is 'dump.' Once a PIP receives a dump request, it adds the requesting PIP into the known PIP list and dumps its knowledge base to the requester. It is the job of the requester to analyze the content and change its own configuration accordingly. Each PIP periodically probes other selected PIPs in the known PIP list. The PIPs probed are chosen according to the structure of the P2P network; one box per subnet is arbitrarily chosen. If a PIP cannot be contacted in a specified period of time, it is removed from the neighbor/known PIP list.

The freshness of information is controlled by versions consisting of time stamps. Each piece of information is stamped with the clock time of the change observer (which might not be a local peer).

Policy Change Planning

Each PIP records its own decisions in a low-level operational policy file. When another PIP appears on the same subnet, it might take over some tasks because of performance or other reasons, and mark the operational low-level policy accordingly. The functional states of PIPs in a particular subnet are managed by a "boss" PIP, whose identity is determined by a booting race condition. Only the "boss" of a subnet can change the low-level behavioral attributes related to that subnet. The "boss" effectively acts as a coordinator to prevent write conflicts.

We must assure that policy changes propagate to every PIP, and that there is a global convergence among the PIPs to one coherent overall policy. This means that all the active PIPs in the IP address closure either accept or reject a high-level policy together. Before a high-level policy is used, a policy proposal is published to the closure. Then the PIPs decide whether the proposal is feasible. Our research does not focus on how to quickly reach consensus in a distributed environment; we choose a simple two-phase protocol and leave the problem of optimizing this protocol to future work. The good news is that because the IP address closure is operating in a controlled environment, the complexity of the consensus problem is significantly reduced.

A high-level policy proposal is accepted only when all active PIPs vote 'yes,' which indicates that all the preconditions in this policy that are related to a particular PIP are satisfied. It is possible that a change is rejected and a PIP votes 'no.' This happens when the physical constraints of the policy are violated. For example, one fixed mapping in the policy file might be impossible because the host is physically on a different subnet than the policy expects.

Self-healing and Failover

While the self-healing features are implemented by redundancy, there are some special considerations for self-healing of DHCP and DNS servers. In particular, any failover should allow for:

1. Seamless management of leases granted by the failed peer.
2. Address spoofing of failed DNS servers during failover.

In our prototype, we handled the first condition but not the second; it is reserved for future work.

In order to provide failover DHCP service from one server to another server, IP leases must be cached somewhere else so they can be managed on a new server. One way to do this is to store the leases in a P2P infrastructure (for example, openDHT [18]). In this way, every IP assignment is recorded in the network, and transitioning from one server to another is easy, because the information is not stored in each individual server alone; replicas are stored in the P2P network. We chose to use the existing DHCP Failover protocol [17], implemented by ISC DHCP. This failover protocol meets most of our goals but has a constraint that it only supports failover in pairs. This constraint limits the number of backup servers to one at any given time. Redundant backup servers are on standby, awaiting future need.

Failures could be in hardware, network, software, etc. The goal of redundancy is to keep the DHCP and DNS service running whenever possible. If a server starts functioning again after a failure, it should be able to recover from the failure; if it fails permanently, the service should still be provided if possible. In the current failover protocol, if a failover pair cannot communicate with each other, they split and share the IP pool for new IP assignment until the communication recovers, because a PIP does not know whether the failure is due to network failure or node failure. If the network partitions and both primary and secondary DHCP servers are assigning IP addresses without splitting, there may be conflicts when a PIP rejoins the network after a long absence. Currently, we are satisfied with the solution of notifying system administrators when the failover mechanism is invoked. If human administrators determine that one of the servers has indeed failed, a backup server can be added to the subnet.

Bootstrapping a PIP

Each PIP acts as a primary DHCP server, secondary DHCP server, or backup DHCP server. A booting state diagram (Figure 4) shows the states of a PIP when it boots. The booted PIP can be in several states depending on the network environment. If it obtains an IP address from a DHCP server, it will enter the 'cloning' state, in which policies are dynamically kept synchronized with the current segment server. If it does not receive a DHCP response and discovers its own IP number from its environment and policy, it will assume that it is alone on the network segment and go into active service as a DHCP server. Else, if a PIP cannot determine its IP address by any means, the boot process fails.

During bootstrapping, a PIP must determine the segment into which it has been plugged. It first sends a DHCP request message on the segment, hoping a DHCP server will respond and assign it an IP address. If not, it probes the network and determines its location, and then assigns itself an IP address based upon that probe.

How does a PIP determine its own IP address if DHCP is not yet running and it is the first potential server? Ideally, we should be able to obtain this location information from lower-level closures – for example, through a broadcast-based protocol. Without such a luxury, we must probe for an IP address that we can

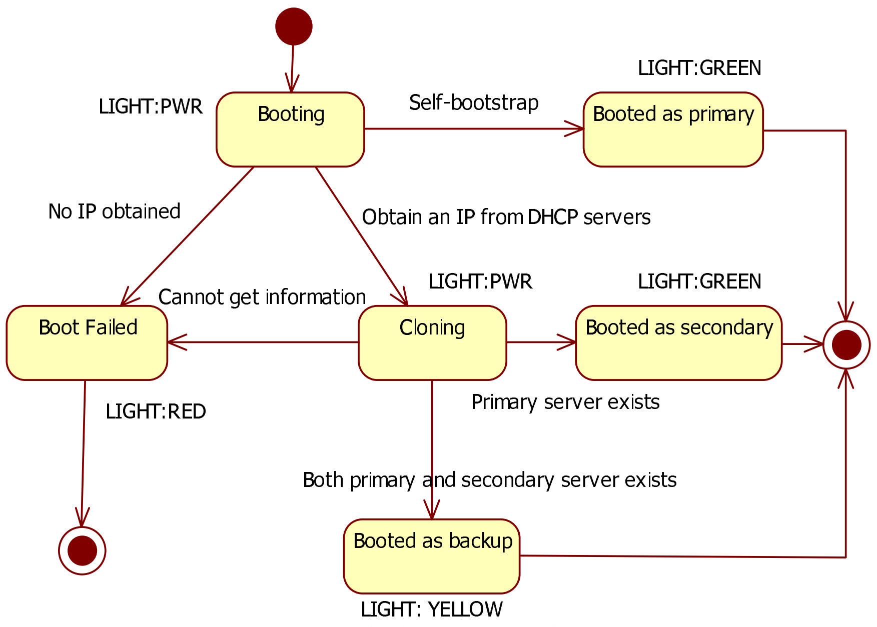

Figure 4: The booting state diagram.

use to exchange information with other nodes. We implemented the probe mode in our prototype. For example, we have the following definition for segment 192.168.3.0/24 in the seed file:

```
<seed>
   <segment>
     <network>192.168.3.0</network>
     <cidr>24</cidr>
     <router>192.168.3.1</router>
     <bootip>192.168.3.2</bootip>
   </segment> ...
</seed>
```

Each node actively probes to determine which segment in the list of possible segments is directly connected to it. The seed file contains a list of primary routers, with one unused IP address (called the bootstrap IP) for each segment. A PIP iterates through the segments and try to ARP the corresponding primary router. If it receives an ARP reply from the router within a specified period of time, then it concludes that it is connected to the corresponding subnet.

In using the probe protocol, a race condition can occur when two PIPs are bootstrapping on the same segment simultaneously. Then both PIPs try to use the same IP address. To avoid the race condition, each node sends an ARP request to resolve the bootstrap IP address. We refer this kind of ARP request as a *claiming ARP*, because the goal of this ARP request is to claim that a node is going to use the bootstrap IP address. If this IP address is already used, the node will receive an ARP reply from the bootstrap IP address, indicating that this address is already in use by another host. Then the booting node will simply abort the bootstrapping process.

If, after a period of time, no other claiming ARP request for the bootstrap IP address is received, the PIP will assign itself that address (we will call this

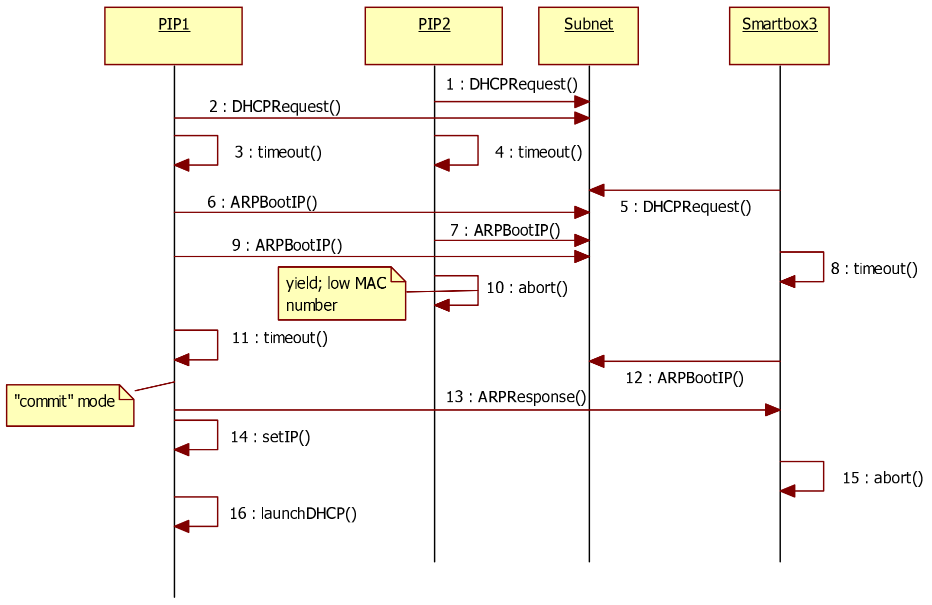

Figure 5: The bootstrapping sequence diagram. PIP1 wins in the bootstrap competition.

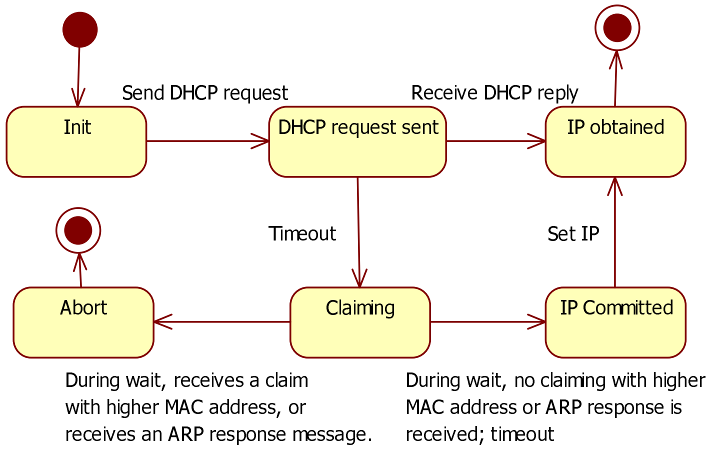
Send DHCP request
Receive DHCP reply
Init
DHCP request sent
IP obtained
Timeout
Set IP
Abort
Claiming
IP Committed
During wait, receives a claim with higher MAC address, or receives an ARP response message.
During wait, no claiming with higher MAC address or ARP response is received; timeout

Figure 6: The bootstrapping state diagram.

state ‘committed’). If before this timeout (commit) event, more than one PIP is booted at the same time, each will receive the claiming ARP request at roughly the same time; the winner is determined based on MAC address. The PIP with a higher MAC address proceeds while PIPs with lower MAC addresses yield quietly. However, if one PIP has already committed its IP address, it will send a ARP reply claiming the IP/MAC mapping, as if it is already using that IP address. Any PIP, even though it may have a higher MAC address, will yield when it receives such an ARP reply, because that means the IP has been taken.

Before the IP address on the PIP is committed, the bootstrapping program is responsible for sending ARP responses so other nodes will yield. After the IP address is set, the ARP response will be generated by the OS. In this bootstrap protocol, the timeout period must be long enough to guarantee the ARP response is received if there is another host using the same IP address. Note that in this protocol, no incorrect ARP replies are sent to the network, so no ARP poisoning is caused by our protocol. Figure 5 shows a sequence diagram of three PIPs trying to boot at about the same time. A state diagram (Figure 6) shows the state transitions for this protocol.

Seed File Distribution

To minimize the work of the system administrator, we designed a mechanism to help with distribution of the seed file. We achieve this goal via a seed PIP, which is a delta PIP, indicating that it moves between subnets to gather information information from PIPs in each one. The seed PIP first self-bootstraps, then provides DHCP service. When a second PIP is plugged into the network, it gets an IP address via DHCP from the seed PIP. Then it configures itself as a failover for the seed PIP. In turn, the seed PIP can be removed without affecting the service, and moved to another subnet, where the process repeats.

Once the seed files are copied, a seed PIP is no different from other PIPs. The administrator can unplug any of the PIPs on the current net and use it as a seed PIP on a different subnet. We intend for the PIP to eventually have a light weight and a small size, so it can be carried easily around to seed other PIPs, e.g., behind firewalls.

The State Transition Problem

A system is rarely static. During its lifecycle, humans will request many changes in system behavior. System administrators need to be able to move the system from one operational state to another. This is called the *state transition problem*.

Traditionally, humans have been in charge of state transitions. The human administrator manipulates each device (in some way) into a new state. When systems become self-managing, however, it is possible for the systems themselves to take an active role in changing states. The ideal situation occurs when the system being self-managed knows the best possible way to change state, so that it serves as a ‘‘co-pilot’’ or ‘‘partner’’ to the system administrator requesting the change.

In autonomic computing, several change planning systems have been developed – for example, CHAMP [9]. CHAMP is a task scheduling system that tries to optimize task scheduling based on cost and time constraints. CHAMP differs markedly from the IP address closure. It tries to solve the scheduling problem for changes so that downtime and disruption are minimized, and distributes the tasks for parallelism; and the calculation is centralized on a single master host. By contrast, the IP address closure does not compute change schedules centrally. Change schedules for the IP address closure can be computed locally, because the IP assignments for different subnets do not depend on one another.

Using the Closure

This section describes how one uses the closure. Low-level closures such as this one pose unique challenges for the system administrator. For the IP address closure to be functional, the system administrator must synthesize a description of its operating environment as well as its operating policies. There is an intimate relationship between contents of the closure configuration file and the routing architecture of the site. Thus the human administrator is in no sense obsolete; changes in the environment must be made known to the closure on a continuing basis.

Our IP address closure’s input is a policy file describing the desired relationships between IP numbers, network names, MAC addresses, and subnets. For example, it specifies which subnets are present and their numbering ranges. Some of this information would ideally be determined by a lower-level routing closure, e.g., the addresses of subnet gateways; here we (temporarily) encode that information into a seed file instead.

When using the IP address closure, the only thing a system administrator must specify is the intended behavior of the IP space; one is relieved from managing superfluous and ‘‘incidental complexity’’ with no behavioral impact [3]. For example, the tedious task of insuring agreement between DHCP servers on the location of routing gateways is managed by the closures and the human administrator need not participate.

The Policy File

In the IP address closure, there are two levels of policy. The first is a high-level policy that defines the overall behavior of the closure and reflects the IP scheme of the whole organization. This is determined by the system administrator. The second is a low-level policy that describes the behavior of the running system and how actual configuration files are generated. This is determined by the closure itself. For example, the number of hosts allowed in a certain subnet is part of a high-level policy, whereas which host serves as primary and which serves as secondary failover server is part of a low-level policy. The high-level policy file contains the DHCP pools of available public IP addresses and private IP addresses, physical subnets, lease period, and some strategies about how the IP address closure is formed. These attributes define the behavior of the closure. The high-level policy specifies the goals of a bootstrap, while the low-level policy represents a steady operating state in which the bootstrap has been accomplished.

The high-level policy file reflects the IP scheme of the whole organization. Some part of the high-level policy may not be realized by a particular closure. Before a new version is released, it can be validated by several rules, including checks for consistency, IP overlapping, and syntax errors. After validation, a new policy will be broadcast to all the servers in the closure. The following code shows an example high-level policy file.

```
<policy ts="1136488200">
  <!-- static mapping from MAC to
                             IP address -->
  <include tag="mac-to-IP">
                       fixed-ip.xml</include>
 <!-- static mapping from MAC to
                              host name -->
   <include tag="mac-to-name">
   fixed-name.xml
   </include>
   <!-- will be maintained by a
                         router closure -->
   <topology>
      <!-- defines subnets connected
                    by DHCP relay agents -->
      <relayed-subnet id="department1">
         <subnet>192.168.1.0</subnet>
         <subnet>192.168.5.0</subnet>
       </relayed-subnet>
   </topology>
   <pools>
      <pool access="private">
         <from>192.168.0.0</from>
         <to>192.168.254.0</to>
         <cidr>24</cidr>
         <max-lease-time>51000
                      </max-lease-time>
         <subpool>
            <from>192.168.3.10</from>
            <to>192.168.3.254</to>
            <cidr>24</cidr>
            <max-lease-time>510000
                      </max-lease-time>
            <include tag="restriction">
                      res.xml</include>
          </subpool>
      </pool>
   </pools>
   <!-- Special rules (exceptions)
           to the previous rules -->
   <!-- Some rejected hosts -->
   <include tag="rejected-hosts">
             blacklist.xml</include>
   <!-- Some VIP hosts -->
    <include tag="VIP">vip.xml</include>
</policy>
```

The high-level policy file does not depict which server is currently serving which subnet, and where the configuration files are located, etc. This type of unnecessary information is part of the "incidental complexity" that closures are designed to avoid. By excluding nonessential and architecture-specific information, the high-level policy can achieve a high level of reusability.

The low-level policy file contains nearly the same information as the DHCP/DNS configuration files, but it also contains the running state of the peer-peer system. For example, the following code is a part of the low-level policy. The 'auth' attribute records the current "boss" in charge of this segment. The 'failover' attribute shows that the failover is on. This protocol distinguishes between owners of information at a relatively fine grain.

```
<closure ts="1136491620">
   <dns>
      <ip>192.168.0.100/24</ip>
   </dns>
   ...
   <subnet-segments>
   <!-- The auth attribute hold the
     current owner of this subnet. -->
   <subnet physical="192.168.0.0"
         authMAC="00:02:3F:1F:9C:88"
         auth="192.168.0.21/24"
         failover="on">
      <id>192.168.0.0</id>
      <netmask>255.255.255.0</netmask>
      <max-lease-time>51000
                     </max-lease-time>
      <pool>
      ...
</closure>
```

When changes are needed, such as changing the range of available IP addresses, or IP renumbering [11], IP address closure can ease the job of an administrator. Currently renumbering is very labor-intensive and requires a series of carefully orchestrated steps. Given a change in policy, the closure could in principle take over this orchestration and accomplish the renumbering with minimal outside help. This includes validating that the renumbering is possible, and actually performing the renumbering once it is proved to be valid, leading the human administrator through a series of foolproof steps.

Implementation Details

We implemented a prototype of the IP address closure using the ISC DHCP [8] and BIND [7] software. The gossip protocol is built on TCP, and policy content is encoded in XML. The information is managed using Berkeley DB XML by sleepycat [21]. Our test environment consists of ten PCs running Linux. They are separated into four IP subnets connected by PCs configured as routers.

To implement self-bootstrapping, we extended the function of DHCP client and implemented the logic shown in Figure 5. The PIP box is pre-installed with the modified version of the ISC DHCP v3.0.2 package. When a PIP is booted, the Ethernet interface is configured to obtain its IP via DHCP. If an IP is obtained from fellow PIPs, the booting PIP will launch the gossiping process; otherwise, the self-bootstrapping process starts. The gossiping protocol is implemented as a pull-only peer-to-peer gossiping application. The transformation from a low-level policy to the actual configuration file utilizes XSLT [22] technology.

In our current setting, the size of contents in a PIP is around 4KB. The cloning of whole contents (from one PIP to a newly installed one) happens in about one second. We set the interval between two pull operations to be 20 seconds. Because of the size of our testing environment, the propagation delay is bounded to 20 seconds as well. Propagation delay is affected by both the frequency of pulling and the number of neighbors each PIP has. In our setting, it is safe for a PIP to have 10 neighbors. It will be interesting to validate this light-weight protocol in a real large-scale enterprise environment, and discover a range of optimal number of neighbors that each PIP should have.

The current capabilities of this prototype are bootstrapping, the dissemination of high-level policy and proposal through a P2P network, high-level to low-level policy translation, and automatic DHCP server configuration update (ISC DHCP only). Future self-managing features of an IP address closure (yet to be implemented) include policy-environment conflict detection, IP address pool shortage warning and auto-allocation. We achieved many of our goals in this prototype: to validate the feasibility of (1) self-bootstrapping, and (2) realizing a distributed configuration based on a high-level policy to provide a robust IP infrastructure.

Autonomics and System Administration

The popular vision of "autonomic computing" (or "self-managing systems") is that there will be no system administrators and systems will manage themselves. This vision is inaccurate and naive. Before an autonomic system becomes functional, much initial setup work must be completed by administrators. After the system is successfully configured into a functioning state, the system is monitored by both self-managing components and system administrators. If a problem occurs and it is beyond the self-healing ability of the autonomic system to correct itself, administrators must take over and restore the system to a functional state.

A rather obvious property of autonomic systems is also their most major pitfall. Current autonomic systems can only cope with predictable failure modes. If something unpredictable happens, a human is required to intervene and take appropriate action. The system can "learn" (postmortem) what it should have done, but cannot cope with the new problem without guidance and help.

Here there is a major and unexpected pitfall for the system administrator. The problems with which an autonomic system cannot cope are also problems that may stump the experienced system administrator. The autonomic system is best thought of as a "junior system administrator" armed with a set of "best practice" scripts that can solve most problems. When a problem does not fit any known description, then by nature, *advanced intervention* is needed. The system administrator who can cope with problems of this nature must be *better* trained than many current system administrators, with an emphasis on efficient and rational troubleshooting. But how (in the context of self-managing closures) does the system administrator achieve this high level of training, when the closure is trying to take control away, and isolate the system administrator from the behavior of the system? One cannot both train and isolate the system administrator. This is a major quandary in the design of autonomic systems: how will the administrator achieve the level of knowledge required to cope with contingencies?

Administering the IP Address Closure

We use the IP address closure as an example to discuss the impact of similar autonomic systems upon system administrators. The system administrators delegate some low level decisions to the closure. Thus, they can focus on the larger picture of IP address assignment schemes. The IP address closure relieves system administrators from the job of backing up policies, because PIPs clone policies from one another and are in essence self-preserving.

However, in no way is the system administrator redundant in the IP address closure. The closure cannot control or define the physical connectivity between devices, or guarantee the architecture of physical or virtual subnets. The system administrator has a permanent role in matching the physical architecture of the network with policies, and in intervening when the closure discovers a mismatch between the physical network and desired operating characteristics.

Another unavoidable role is that of bootstrapping the system from a non-functional state to a self-managing state. In our closure, this is accomplished by physical moves of devices. This eliminates common human errors in copying configurations and makes the bootstrapping protocol more or less foolproof, but requires a basic understanding of how the PIPs self-replicate.

Lessons Learned

The PIPs show us something fundamental about the ongoing relationship between system administrators and autonomic elements. The administrator is far from obsolete, but also somewhat removed from the day-to-day management tasks that were formerly part of the job. The system administrator becomes a crafter of policies, rather than just a troubleshooter. Far from being less skilled, the system administrator of the closure system actually needs a higher level of sophistication to deal with unexpected problems.

The changing role of system administrators includes the deployment and bootstrapping of autonomic systems. Each autonomic system has a set of preconditions that must be met before it can function as designed. System administrators must maintain the appropriate environment for the autonomic system. Before the autonomic mechanisms are implemented from top to bottom, each autonomic system must rely on human administrators to cope with the bottom layers. Although these autonomic systems provide self-configuration features, deployment and bootstrapping are unavoidable and uniquely human tasks.

The role of system administrators also include tuning and validation of the new autonomic systems. Many autonomic systems contain heuristics that require the collection and analysis of real production data. Before the system is tuned, system administrators may have to manage the system manually. After the system is tuned, it must be validated to make sure that it is configured as desired.

Administrators also must intervene when a problem cannot be handled by an autonomic system. This poses new challenges to autonomic systems and their users. Unlike current practice in which the administrators have absolute control, system administrators must turn off certain parts of the automated process and take over, just like taking over from automobile cruise control or auto-piloting. The responsibilities must be well-defined and documented. The switching process between autonomic and manual management must be well documented and practiced.

An autonomic system itself is more complex than a corresponding traditional system. For example, in configuring our PIPs, one must describe their operating environment and policies in great detail, giving information that would not be known by hosts in a non-autonomic DHCP/DNS infrastructure, such as the locations of routers on foreign subnets. This extra configuration, however, pays off when the resulting fabric of servers relieves one from routine tasks such as rebuilding servers or propagating changes to the network.

One primary obstacle to acceptance of autonomic solutions is trust [1, 10]. People often do not trust that machines can handle tasks reliably, when humans will lose control. In truth, autonomic solutions are "assistants" rather than masters; the fabric of management still contains both machines and humans. This paradigm is especially necessary at the lower levels, where human assistance is required. The human administrators can use help in implementing complex processes. One goal for autonomic systems is to automate IT service and resource management best practices [4]. Automating these best practices can best gain the trust of the management and administrators. Further, autonomic assistants can help humans track the state of a task. Our problem domain is close to the hardware; so close that a human element cannot be avoided to serve as the "hands and feet" of the system. Accountability issues are also unavoidable. Who should be responsible if a system is not tuned well and does not meet the specific requirement of a site and cause downtime?

Our brief discussion of the changing role of system administrator may seem daunting, but the job is in no danger of extinction. Current closures require extensive bootstrapping and handling of contingencies, and require monitoring by a highly skilled system administrator. In fact, management of autonomic systems seems to elevate the profession in several ways:

1. by requiring a high level of system administration expertise.
2. by redefining the role of system administrator as someone who directly interacts with policy.
3. by providing (through interaction with policy) upward mobility to management positions.
4. by providing a much needed human interface between autonomic elements and upper management.

Exception Handling

The effectiveness of an autonomic solution depends upon the efficiency with which humans can communicate with it. Our PIPs cannot solve all problems, so that their ability to effectively communicate problems to humans is crucial to their success.

A key feature of any autonomic system is how it handles cases in which self-management does not resolve an issue. The goal of exception handling in autonomic systems is to report any violations of policy and resolve them. For example, in the IP address closure, suppose an interface is declared to have a fixed IP address. When an administrator activates that interface, if this host is not physically located on the same subnet as it should be, the interface might get an IP address from DHCP on a different subnet than desired. In this scenario, no obvious error is generated. However, this is a clear exception to the policy. In this example, the root cause is that the interface is not connected physically to the proper subnet. We may choose to correct the root cause by physically moving the interface or perhaps setting up a VLAN to simulate physical rewiring. Alternatively, we could choose to reject the policy on the grounds that it is not implementable. This constitutes a form of *exception* in the closure.

The concept of exception has been widely used in the computer science field. Exceptions have been used mostly to express a predictable abnormal scenario that can be categorized in advance. Researchers have explored exception handling mechanisms in both intra-component and inter-component situations. For example, Romanovsky divided exception handling into local error detection in one component and coordinated action level handling among components [19]. Here we focus on handling of unexpected exceptions, or exceptions with unknown causes.

Because a closure defines observable behaviors, it is natural to define the possible exceptions raised by this closure also in terms of observable behaviors, just like the symptoms of patients. However, unlike the policy file, the more detailed the exception, the more helpful the information. A simple exception containing no extra information will almost definitely require a human to troubleshoot the problem.

The exceptions that a closure could raise can be divided into two categories:

- Exceptions that cannot be handled by this closure. The cause of the exception is out of the scope of control of a certain closure. i.e., self-healing does not function properly in one situation. For example, in the Apache closure, if the file system is corrupted, the closure cannot possibly work properly, thus an exception must be thrown. Sometimes, the reason for the exception is unclear. Thus a generic exception might be raised.
- Exceptions that may be handled by this closure but that the closure chooses not to handle. Rather, the closure wants the exception to be handled by other closures.

For example, in the Apache closure, if the server is unstable, the closure may choose to restart the server. Or, if the closure decides that a better way to handle this is to reboot the whole host machine, it may raise an exception and let another entity handle it (such as a host closure).

A special type of exception is related to human input. When the closure discovers that the intention of the administrator is unclear, or it encounters a condition where more human input is needed, it should raise an exception to request more information, instead of relying upon itself.

Exceptions can be handled by other closures or human administrators. Since we cannot wait to have closures to be built on all the layers and switched on at once, it is necessary to have a way for closures to request services from other non-closure systems or human administrators. In the exception-handling process, after the event causing the exception is resolved, the closure can be contacted manually or programmatically to continue its work. In a true autonomic system, most exceptions should be handled by a program, rather than by a human administrator.

Conclusions and Future Work

We propose an "IP address closure," a self-managing IP management infrastructure providing DHCP and DNS services. The IP address closure mimics the best practices that administrators discovered in practice, and automates them through the coordination among Peered IP management nodes (PIPs). Thus, the IP address closure is designed to gain the trust of the system administrators to assist with their work.

The task of making low-level systems self-managing still requires solving many open problems. The key problem for IP management is to maintain an effective interface between the fabric and its human counterparts. Human administrators are not obsolete, and they are still critical because autonomic systems cannot escape exception problems due to physical limits upon architecture. However, designing policies and resolving exceptions might require a new set of skills for existing administrators. The policy still depends upon architecture.

The most complex and challenging problem is that of planning for safety in very complex changes. When policies change, there are often safe and unsafe ways to transition between policies, where an unsafe transition is one that temporarily exposes a security risk.

Another related problem is how to make the lower layers (routing and switching) self-managed in a similar way. These layers suffer from the same "bootstrap problem" that we observe for IP address management; the management fabric has to use what it manages for its own sustenance, and cannot do that until it manages that fabric. The simple solution of managing routing via an out-of-band management network may not be cost-effective for many sites.

Clearly, there are many issues to explore. If there is a single most important contribution of this paper, it is that the closure idea is possible at the IP layer, and that – even with bootstrapping difficulties – self-managing fabrics can function near the physical layer of a network, provided that there is a carefully orchestrated relationship between the self-managing fabric and its human partners.

The role of administrators in the autonomic era has already changed. Instead of being obsolete, autonomic systems challenge system administrators to obtain a higher level of expertise, including knowledge of policy design and architecture, tuning, and troubleshooting. At the same time, autonomic systems elevate the system administration profession and shorten the distance between management and system administration through the common language of policy-based interfaces. Some system administration jobs may be lost to autonomic systems, but those that remain may well enjoy better advancement opportunities, as well as increased respect and recognition for the profession.

Author Biographies

Ning Wu is pursuing his Ph.D. at Tufts University. His research interests are in system management, autonomic computing, system integration, and P2P systems. Before studying at Tufts, he had worked as an engineer for Genuity and Level 3 Communications Inc. He received an M.S. from State University of New York at Albany, an M.E. from East China Institute of Computer Technology, and a B.S. from Southeast University in China. Ning can be reached via email at ningwu@cs.tufts.edu .

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M.I.T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Computer Science at Tufts. Prof. Couch is the author of several software systems for visualization and system administration, including Seecube(1987), Seeplex(1990), Slink(1996), Distr(1997), and Babble(2000). He can be reached by surface mail at the Department of Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@cs.tufts.edu .

Bibliography

[1] Chan, Hoi, Alla Segal, Bill Arnold, and Ian Whalley, "How can we trust an autonomic system to make the best decision?" *2nd International Conference on Autonomic Computing (ICAC 2005)*, pp. 351-352, 2005.

[2] Cisco Systems, *Cisco CNS Network Registrar Users Guide, Software Release 6.1*, 2004.

[3] Couch, Alva, John Hart, Elizabeth G. Idhaw, and Dominic Kallas, "Seeking closure in an open world: A behavioral agent approach to configuration management," *Proceedings of the 17th Conference on Systems Administration (LISA 2003)*, pages 125-148, 2003.

[4] Ganek, A. G. and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal*, Vol. 42, Num. 1, pp. 5-18, 2003.

[5] IBM, *An architectural blueprint for autonomic computing*, IBM white paper, April, 2003.

[6] International Network Services, *IPControl*, http://www.ins.com/software/ipcontrol.asp .

[7] Internet Systems Consortium, Inc., *ISC BIND*, http://www.isc.org/index.pl?/sw/bind/ .

[8] Internet Systems Consortium, Inc., *ISC Dynamic Host Configuration Protocol (DHCP)*, http://www.isc.org/index.pl?/sw/dhcp/ .

[9] Keller, A., J. Hellerstein, J.L. Wolf, K. Wu, and V. Krishnan, "The champs system: Change management with planning and scheduling," *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, Kluwer Academic Publishers, April, 2004.

[10] Kephart, Jeffrey O. and David M. Chess, "The vision of autonomic computing," *IEEE Computer magazine*, January, 2003.

[11] Limoncelli, Tom, Tom Reingold, Ravi Narayan, and Ralph Loura, "Creating a network for lucent bell labs research south," *Proceedings of the 11th Conference on Systems Administration (LISA 1997)*, pp. 123-140, 1997.

[12] Logan, Mark, Matthias Felleisen, and David Blank-Edelman, "Environmental acquisition in network management," *Proceedings of the 16th Conference on Systems Administration (LISA 2002)*, pp. 175-184, 2002.

[13] Lucent, *Lucent network management software for enterprises*.

[14] Melcher, Brian and Bradley Mitchell, "Towards an autonomic framework: Self-configuring network services and developing autonomic applications," *Intel Technology Journal*, Vol. 8, Num. 4, Nov., 2004.

[15] Mockapetris, P., "Domain names – concepts and facilities," *RFC 1034*, 1987.

[16] Mockapetris, P., "Domain names – implementation and specification," *RFC 1035*, 1987.

[17] Network Working Group, *DHCP failover protocol*, 2003, http://www3.ietf.org/proceedings/04mar/I-D/draft-ietf-dhc-failover-12.txt .

[18] Rhea, Sean, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu, "OpenDHT: A public DHT service and its uses," *Proceedings of ACM SIGCOMM 2005*, 2005.

[19] Romanovsky, A., "Exception handling in component-based system development," *The 15th Int. Computer Software and Application Conference, COMPSAC 2001*, 2001.

[20] Schwartzberg, Steven and Alva Couch, "Experience in implementing a web service closure," *Proceedings of the 18th Conference on Systems Administration (LISA 2004)*, 2004.

[21] Sleepycat Software, *Berkeley DB XML*, http://www.sleepycat.com/products/bdbxml.html .

[22] W3C, *XSL Transformations (XSLT) Version 1.0*, http://www.w3.org/TR/xslt .

[23] W3C, *Web services architecture*, 2004, http://www.w3.org/TR/ws-arch/ .