17th ICCRTS
"Operationalizing C2 Agility"

Dynamic Discovery and Configuration in Airborne Networks

Primary Topic: Networks and Networking
Alternate Topics: Cyberspace Management; Architectures; Technologies and Tools

N. J. Charbonneau, J. D'Amelia, E. G. Idhaw
The MITRE Corporation
202 Burlington Road
Bedford MA

Point of Contact: Neal Charbonneau
charbonneau@mitre.org /(781) 271-5065

The MITRE Corporation
202 Burlington Road
Bedford MA

Dynamic Discovery and Configuration in Airborne Networks

Neal Charbonneau, Jeff D'Amelia, Elizabeth Idhaw
The MITRE Corporation
202 Burlington Rd.
Bedford, MA

***Abstract*—Tactical networks require a significant amount of pre-planning and configuration before deployment. This process is error-prone, time consuming, and ultimately reduces mission flexibility. In this paper we describe an approach to node discovery and automated configuration for dynamic tactical networks. Our architecture allows unplanned nodes to join a network dynamically with little pre-planning or operator intervention. Once joined, we can distribute configuration changes to any node in the network. By introducing our neighbor discovery protocol and configuration dissemination capability, we can reduce the time it takes to integrate tactical nodes into the network and limit the possibility of mis-configuration while improving mission flexibility.**

***Index Terms*—auto-discovery, configuration, network-management, tactical networks**

I. Introduction

The current state of the practice in deploying tactical networks requires configuration of the network infrastructure to occur prior to deployment. This pre-planning stage takes a significant amount of time and effort, and as a result it is difficult for unplanned nodes to later join the network. For example, to have a new node join the network, a network operator must manually configure the networking equipment by dictating low-level device configuration commands to someone on the platform. This process is error-prone and time-consuming, particularly since the commands may be entered by someone with limited networking expertise. In addition to the difficulty in adding new nodes, there are no solutions for efficiently distributing configurations to devices in the network to support changes in the mission. These factors lead to rigid and inflexible networks, which impact the higher layers and limit the capabilities that the network can support.

To make tactical networks more flexible, we propose two main capabilities. The first is **node discovery**. The network must be able to dynamically discover new, possibly unconfigured, nodes as they become capable of joining the network (i.e., they come within radio range).The second is **configuration dissemination**. There must be a mechanism to distribute configurations to new nodes and to existing nodes. With these two capabilities, the network can be changed dynamically and with little human intervention.

Both problems in tactical environments are unsolved by commercial products for a number of reasons. As we will discuss in Section II, the underlying network links may actually consist of black-box proprietary radio networks creating additional IP network hops. Traditional neighbor discovery approaches do not work in this environment. Another important issue for both discovery and configuration is that most solutions from industry are vendor specific, meaning they do not support heterogeneity of networking devices. Lastly, commercial networks are typically static and change on much longer timescales than tactical networks, so there is little incentive for industry to work on these problems. We will discuss related work in more detail in Section IV, but we found that this problem requires novel solutions.

We propose a protocol that allows nodes to dynamically discover their neighbors and establish initial connectivity (Section III-A). Then we propose a mechanism that allows configurations to be disseminated across the network. Finally, we propose a device-independent configuration format that allows nodes to automatically apply configurations (Section III-B). This approach solves the two problems previously discussed. The protocol can be used to discover new nodes and push configuration to them, allowing them to join the network automatically with little or no human intervention. It can also be used to disseminate configuration data throughout the entire network to change other devices over the

course of the mission. This capability allows operators to dynamically adjust for unexpected mission variations, perform network administration, and improve network performance. This can be done by an experienced operator and automatically distributed/applied throughout the network without any other humans in the loop.

The paper is organized as follows. We discuss the technical challenges in more detail in Section II. We present our solution in Section III. We then discuss related work in Section IV. Lastly, we conclude in Section V.

II. Problem Definition and Assumptions

In this section we discuss in more detail the technical problems of dynamic node discovery and configuration dissemination in tactical networks. We begin by providing definitions.

We assume that each node in the network that we wish to configure is a **platform** (e.g. an aircraft). A platform has one or more networking **devices**, such as routers, switches, performance enhancing proxies (PEPs), etc. The devices on a platform may be supplied by different vendors running different software. For example, Cisco devices run Cisco IOS while Juniper devices may run Junos. We will use the term node and platform interchangeably. There is at least one router on each platform that is considered the **controlling device**. This device handles all traffic entering and leaving the platform. The protocol will run on the controlling device of each platform. Any node that a platform discovers with the discovery protocol is considered a **neighbor**. **Configurations** specify behavior for a particular device and are disseminated throughout the network. For example, a configuration can enable Open Shortest Path First (OSPF) routing on a particular interface of a device. The configuration contains a collection of generic device commands, but will ultimately be translated to device specific commands (e.g., a sequence of Cisco IOS commands).

The communications links between these platforms may consist of line-of-sight (LOS) tactical radio terminals, beyond-line-of-sight (BLOS) satellite terminals, or a collection of both. The communications links themselves may actually be multi-hop networks. For example, in Fig. 1, the path from the radio terminal connected at Platform A to a radio terminal at Platform D may actually have to go through the radio terminal at Platform B first. The routers at Platform A and D would not be able to determine that they are connected by a multi-hop network. The routers only communicate with their attached radio terminals, which may or may not be in direct communication with each other.

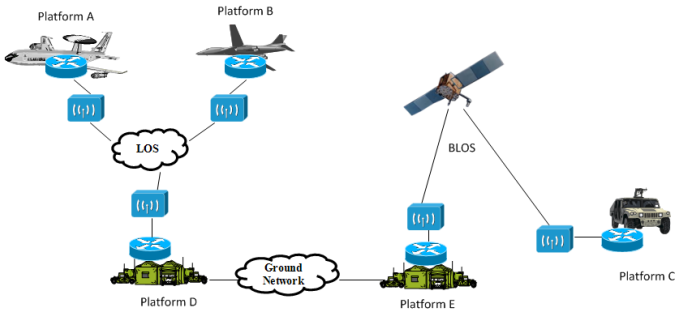

Fig. 1. A sample network with a variety of platforms and communication links. If Platform B is an unplanned node, it could be discovered by Platform A or D and retrieve configuration data in order to join the network.

We use Fig. 1 as an example of the types of networks considered for this work. The figure shows a variety of platforms and communication links. We only show one router on each platform, but the platform could consist of multiple devices. In this example Platforms A, B, and D may be neighbors, as well as D/E and E/C. If a configuration is added to the network at any platform, it will be disseminated to all other platforms.

A. Discovery

We will now discuss in detail the problems with dynamic node discovery in tactical networks. The major technical challenges are as follows:

- Lack of configuration on platforms joining the network prevents access to services available on the network. Because the platform is not properly configured, it is not possible to establish IP connectivity between the platform and other network nodes. This prevents us from using an application-layer solution.
- Lack of configuration means routing protocols cannot discover neighbors. In normal scenarios, once routing protocols are configured, all of the nodes in the network are discovered. Without proper configuration, the protocols cannot talk to each other and will not be able to discover the new node.
- Typically neighbor discovery uses link-local messaging for discovery, but the introduction of IP enabled radio terminals may prevent this. This means we cannot use existing discovery protocols or applications.

While there is significant work on discovery and autoconfiguration in Mobile Ad-hoc Networks (MANET) and discovery protocols, these protocols do not solve

our problem for two main reasons. One is the lack of configuration or the fact that the devices may be configured for different networks. The protocols have to be properly configured before they can work. The other is that the radio clouds may be multi-hop networks which prevent link-local messaging from being used. This constraint is due to particular radio terminals, but because of it we designed our discovery protocol to be able to discover other nodes across arbitrary network topologies, rather than being limited to neighbors at the other end of the link. For more details about the particular environments we consider, see [1].

The goal of our discovery protocol is then to be able to find new (unconfigured/mis-configured) nodes eligible to join the network through an arbitrary number of hops and setup initial connectivity. We call this initial connectivity an **orderwire**. An orderwire allows bidirectional communication to occur between two nodes. With the orderwire established, configuration data can be pushed to the new node so that it can fully join the network.

B. Configuration

The major technical challenges for configuration dissemination are as follows:

- Dissemination should be done in a distributed manner, rather than centralized, to avoid a single point of failure.
- In order to support a heterogeneous vendor network, the configuration format should be device-independent.
- The configuration of devices can be done at two levels of granularity. Some configuration applies at the device level and some configuration applies to individual interfaces. Different devices also require different configurations. This means that the configuration format and dissemination mechanism must be able to specify which configuration snippets get applied to particular devices and interfaces.

We prefer a distributed scheme so that there is no single point of failure and consequently configurations can still be disseminated in the case of temporary network partitions, which could occur frequently in a tactical wireless environment. A configuration can be provided by any node and distributed to all other reachable nodes. This also means that we need the ability to target particular nodes or node types. Lastly, we assume that the network will consist of heterogeneous device types, so the configuration itself should not be specified in device-specific commands.

The goal is to define a flexible configuration format along with an advertisement protocol that can disseminate the configurations throughout the network from any node currently on the network. The configuration format will be device-independent and translated to specific devices as needed.

III. System Architecture

In this section, we present the architecture for dynamic node discovery and configuration. We will discuss the architecture in two separate sections, the first being discovery in Section III-A and the second being configuration in Section III-B.

The protocols we are about to describe use IP multicast to send their messages. Multicast allows a single sender to transmit a message to a group of destinations, rather than a single destination as in traditional unicast. This means that when sending a multicast packet, the destination IP address is not the address of a host, but of a multicast *group*. Any other node on the network can join (or leave) a multicast group to receive messages directed to that group. Multicast has been deployed and demonstrated in similar environments so we assume that all the nodes in the network are multicast-capable. We will explain why we used multicast in the following sections. We also assume that each platform is assigned a globally unique **device ID** used to identify it. This can be generated according to RFC 4122 [2] or, if shorter IDs are required, using a MAC address from the controlling device.

A. Discovery

The discovery protocol uses multicast messages to discover neighboring nodes. Each platform will join the multicast group on each interface of the controlling device. The controlling device then periodically sends out multicast packets we refer to as *beacons*. When a platform receives a beacon packet, it records the interface on which the packet was received. This allows us to determine which neighbors are attached to each interface (rather than just knowing a neighbor exists). Next, a static route is added to the neighbor (we can get the neighbor's IP address from the IP header of the beacon). Once two neighbors have exchanged beacon packets, they will both have added static routes to each other. With the static routes in place, bidirectional data transfer can begin, i.e., the orderwire is established.

The format of the beacon packet is shown in Fig. 2. The beacon packet contains the sender's device ID along with a beacon and stale interval. The beacon interval

Bit offset	0-3	4-7	8-11	12-15
0	Preamble			
16	Version		Type	
32	Device ID			
...				
160	Beacon Interval		Stale Interval	
176	Number of neighbors			
192+	Variable Length Neighbor Data			

Fig. 2. Beacon packet format. The version field denotes the version of the protocol while the type field denotes the time of message: beacon or advertisement.

Algorithm 1 Processing of beacon packets.

```
1:  packet, interface = RecvBeacon()
2:  updateNeighbor(packet.deviceID, packet.src, lastRefresh=now())
3:  if not routeAdded(packet.deviceID) then
4:      addStaticRoute(packet.src, interface)
5:      updateNeighbor(packet.deviceID, packet.src, routeAdded=true)
6:  end if
7:  for neighbor in packet.neighbors do
8:      if neighbor.deviceID == myDeviceID and neighbor.routeAdded == true then
9:          updateNeighbor(packet.deviceID, packet.src, routeToMe=true)
10:     end if
11: end for
```

specifies how often the sender will send a beacon packet. The stale interval is used to determine how long to wait before a neighbor node is timed-out. If a node stops receiving beacon packets from a neighbor for a period longer than the neighbor's stale interval, the node will assume the neighbor is no longer active. The beacon packet also contains all of the discovered neighbors, as well as each neighbor's collection of addresses and whether a static route to each address has been established.

Algorithm 1 shows how a node processes incoming beacon packets. The system will keep state information about discovered neighbors. State is kept for $(\textit{neighbor deviceID}, \textit{neighbor IP})$ pairs since we may hear from the same neighbor (same deviceID) on different interfaces. In the pseudo-code we use an $updateNeighbor$ function to modify this state about neighbors. We also use the notation $packet.X$ to access different fields of the received packet. Line 2 updates the last time a beacon was received. If this becomes larger than the stale interval for that neighbor, the neighbor is pruned. Lines 3- 5 are used to add a static route to a new neighbor. Lines 7- 9 check to see if the neighbor has added a route to this node yet. As we mentioned previously, the beacon sender will include a flag indicating which neighbors the sender has added a route to. Here, the receiver checks to see if the flag is set. This allows two neighbors to determine when they have added routes to each other. Note, the sender must include all neighbors in the beacon packet because the packet is sent only once using multicast and received by all neighbors. Each receiver will then search the list for itself.

At the end of this phase, both neighbors have added routes to one another. This is a requirement for sending configuration data, discussed in the following section. Without both nodes having routes, a regular unicast connection using TCP would fail. The addition of these routes constitutes an **orderwire**, whereby neighbors can exchange configuration information.

The use of multicast allows the system to discover unconfigured neighbors, as long as the new platform joins the multicast group. We note that while we use multicast, the processing of packets once they reach each platform is different from normal IP multicast. As an example, consider the Protocol Independent Multicast (PIM) [3] protocol. When PIM dense-mode is used to route multicast messages, a node forwards an incoming multicast packet only if it is received on the reverse path interface (to prevent loops), otherwise the packet is dropped. In our case, the new unconfigured node will not have any paths so we have to intercept the beacon packets before they are processed normally. We also do not forward beacon packets beyond neighbors. Once a platform receives and processes a beacon packet, the packet is discarded. In this way, the protocol behaves like a routing protocol discovering neighbors, except that it can traverse multiple hops between neighbors.

B. Configuration

In this section we discuss device configuration. First we discuss a multicast protocol, similar to discovery, that is responsible for disseminating the configuration throughout the network. Next we introduce the format that specifies how the configuration files are structured.

Bit offset	0-3	4-7	8-11	12-15
0	Preamble			
16	Version		Type	
32	Device ID			
...				
160	Number of Advertisements			
176	Advertisement 1			
192+	Advertisement n			

Fig. 3. Advertise packet format. The first 20 bytes are the same as the beacon packet.

Lastly, we show how the configurations are applied to devices in the network.

The configurations files that are disseminated across the network are broken up according to device type. For example, there are separate configuration files for routers and for switches. Each configuration file has a version number associated with it that is incremented whenever the configuration changes.

When a configuration changes, it must be redistributed to each node in the network. To make this process more efficient, we use separate configurations for each device type to reduce the file size and limit the number of times the file must be transferred between neighbors. If everything were contained in a single configuration file, the large file would have to be redistributed every time something changed. This would result in a large amount of data being frequently passed between neighbors, even for small changes to a single device. We currently assume there is out-of-band coordination between multiple network operators making changes at the same time in order to prevent versioning conflicts. An automatic approach to this conflict resolution problem is left as future work.

1) Advertisement: The advertisement protocol is used to disseminate a list of available configurations across the network. Once a node receives a configuration, the node advertises the new configuration's details with a multicast protocol to inform its neighbors of the presence of updates to the configuration. The packet format is shown in Fig. 3. The packet is a list of available configurations for different devices. Each listed configuration is a two-tuple consisting of a device type identifier and a version number. Like the aforementioned discovery beacon, the configuration advertisement is transmitted on a periodic basis.

When a node receives a configuration advertisement, it compares the advertised configuration version to its current configuration version. If the advertised version is newer, the node will request the configuration from the neighbor advertising the newest version by issuing a unicast request for the data. Configuration advertisements are not limited to only the configurations used by a specific platform. A platform will request and advertise any configuration, not just the configurations that apply to the devices on the platform. In this way, configurations are disseminated throughout the network hop-by-hop without requiring a central server.

It is important to note that a node will only process configuration advertisements once the node has a route added to the neighbor *and* the neighbor has a route added back to the node. Information about added routes is available via the discovery protocol. Once these conditions are met and it is found that a neighbor is advertising a newer configuration, the node will request the configuration, store the configuration locally, and install the configuration on its local devices. Once this process is complete, the node will begin advertising the new version of the configuration to its neighbors.

2) Format: In this section we present the configuration file format. Because the configuration describes devices network-wide, the format must support targeting specific devices, allowing particular sections of the configuration to only apply to certain devices. To do this, each device can be assigned one or more **roles**. Roles are specified by network operators or users and offer an informative description of a device and its use in the network. For example, a role may classify a platform as an *AirborneNode*. If there is a piece of configuration that specifies all airborne nodes must implement a specific access control list (ACL) on its router, each node assigned this role would change its router's configuration to implement the specified ACL. We denote a piece of configuration for specific targets as a **requirement**. The configuration itself is a list of requirements (see Fig. 5) where each requirement has its own version number. The version number of the entire configuration is incremented whenever one of the requirements changes. This allows devices to determine which requirements have changed between configuration versions and only update those specific requirements.

The configuration within a requirement could be low-level device commands like Cisco IOS, but this would require vendor homogeneity. Instead, we define a set of

```
<OSPFNetArea>
   <Addr>10.8.8.1</Addr>
   <Mask>255.255.0.0</Mask>
   <Id>0</Id>
</OSPFNetArea>
```

(a)

```
<OSPFNetArea>
   <Addr>$ReversePathIP</Addr>
   <Mask>$ReversePathMask</Mask>
   <Id>0</Id>
</OSPFNetArea>
```

(b)

```
#IOS translation
def OSPFNetArea():
    Mask=InverseMaskIOS(Mask)
    WriteCommand("router ospf $OSPFID")
    WriteCommand("network $Addr \
    $Mask area $Id")
    WriteCommand("exit")

#Junos translation
def OSPFNetArea():
    WriteCommand("set protocols ospf \
    area $Id interface $Addr")
```

(c)

Fig. 4. (a) and (b) show XML representations of an abstract command to enable OSPF on the specified network. (b) shows how substitutions can be used, where the receiving node will replace variables starting with '$' with local values. (c) shows a the code that converts the abstract commands to device specific commands.

abstract commands that will get translated to device specific commands. The abstract commands are defined as a set of command names and their parameters, formatted in XML. For example, the *OSPFNetArea* command adds an interface to an OSPF area, as shown in Fig. 4(a). The command has parameters *Addr* (the interface to add), *Mask*, and *Id* (the OSPF area ID). We refer to the parameter/value pair as a binding. For the *OSPFNetArea* command, the *Addr* parameter might be bound to the value *10.8.8.1* as shown in the figure. We are currently working on defining a more formal specification for the abstract commands.

Another feature supported by the configuration format is **substitutions**. Substitutions are used in configuration files to allow the receiver to substitute local values for a generic parameter. This allows a node to advertise a configuration without knowing specific details about device(s) that will receive the configuration. We show an example of substitutions in Fig. 4(b). The values for *ReversePathIP* and *ReversePathMask* will be substituted at the receiver for the IP address and network mask of the interface on which the configuration was received.

Now we will discuss the overall configuration file format with requirements. An example configuration is shown in Fig. 5. The configuration shows three requirements, each having a name, version, and targets. When a platform receives a configuration, it matches the targets with the roles of the devices and selects which requirements must be applied. The actual configuration is specified within the $Config$ XML tags. The configuration is extracted and will be translated to specific commands for that device. In addition to requirements having targets, the configuration snippets can also have targets for interfaces. This means that the configuration snippet will be duplicated for each interface. For example, in Fig. 5 the configuration in the

```
<xml>
  <Req name="MCast" version="1"
       targets="device=*">
    <Config>
      <MulticastRouting/>
    </Config>
  </Req>
  <Req name="OSPF" version="1"
       targets="device=*, interface=*">
    <Config>
      <OSPFRouterID>
        <Id>$ReversePathIP</Id>
      </OSPFRouterID>
    </Config>
    <Config target="*">
      <OSPFNetArea>
        <Addr>$IntIP</Addr>
        <Mask>$IntMask</Mask>
        <Id>0</Id>
      </OSPFNetArea>
    </Config>
  </Req>
  <Req name="Intervals" version="2"
       targets="device=*,interface=Radio">
    <Config target="Radio">
      <OSPFIntervals>
        <Interface>$IntName</Interface>
        <Hello>10</Hello>
        <Dead>20</Dead>
      </OSPFIntervals>
    </Config>
  </Req>
</xml>
```

Fig. 5. Sample configuration file.

⟨*Config target="Radio"*⟩ tag will be applied to all radio interfaces on the device.

3) Translation: When a requirement matches the roles of a particular device, the configuration is extracted for translation. The translation from abstract commands to device commands is done using a subset of the Python language. For each vendor (Cisco IOS, Juniper Junos, etc.), a translation file is supplied with code to do the translations. For each abstract command, there is a function definition. Each function has access to the bindings of that abstract command and some pre-defined functions to output the final configuration. For example, in Fig. 4(a), the bindings that the code has access to would be *Addr* and its value 10.8.8.1, along with *Mask* and *Id* with their values. Example translations to IOS and Junos for the *OSPFNetArea* command are shown in Fig. 4(c). The code has access to all of the bindings (*Addr*, *Mask*, and *Id*) and uses the *WriteCommand* function to output the final configuration. There are a number of helper functions available, including *InverseMaskIOS* as shown in the figure. After translation, the device-specific configuration is sent to the device.

IV. Related Work

There are a number of existing solutions for the dynamic node and service discovery problem. The IETF Zeroconf Working group [4] defined an architecture to allow dynamic service discovery in local area networks using link-local techniques, which do not work in our case. There are proprietary or vendor specific solutions such as Cisco's CDP [5]. A number of MANET routing protocols also allow dynamic discovery of neighboring nodes. We cannot use these common approaches in our work because of the previously discussed issues (see Section II). The radio terminals may introduce multiple hops between any two nodes, which breaks any link-local type discovery. This constraint forces us to develop a new discovery technique.

There is work in network configuration management, but most approaches require a centralized architecture, which is in conflict with our requirement of a distributed system. We have borrowed some ideas from these systems though. For example, roles are similar to classes in [6] and our configuration format support some features of the configuration template language presented in [7]. We also wrote our own translation layer to target multiple vendors. The IETF NETCONF Working Group [8] is proposing a protocol and modeling language for configuration of heterogeneous networking devices, but this work is not widely supported by vendors at this time.

We also considered policy-based network management techniques [9]. Policies are similar to the *requirements* that we propose in this paper. The difference is that policies are typically applied to determine a course of action. For example, a new incoming flow may trigger a policy look-up to determine how to handle the flow. In our work, requirements are automatically applied without any need for policy look-up, policy decision/enforcement points, or other policy based mechanisms. It is also the case that most work in policy-based network management focuses on quality of service (QoS) issues.

V. Conclusion

In this paper we have presented a neighbor discovery protocol that can traverse arbitrary network topologies and establish initial bidirectional connectivity between two network devices (e.g., routers) regardless of their initial configuration. We also propose a configuration mechanism that can disseminate configurations throughout the network and automatically apply configurations without operator intervention. These two capabilities allow unplanned nodes to dynamically join a network with little operator intervention.

In evaluating our approach, we were limited to a qualitative assessment of our prototype, where we compared the ability of a human operator (ranging from novice to experienced) to configure a newly discovered router, with our software's ability to perform the same task automatically. This comparison can be affected by numerous conditions including the difficulty and length of the desired configuration, potential operator input errors, etc. While these limitations prevent us from quantitatively defining the improvement demonstrated by our system in a generic way, our assessment showed a significant improvement in configuration time (at least one order of magnitude) when our system was compared against manual configuration by a set of operators.

We are currently pursuing several extensions to the project. We are working on developing and integrating an artificial intelligence or reasoning-based model that can map high level requirements to abstract device commands. For example, instead of having to specify in the configuration all of the abstract commands to enable OSPF on a set of interfaces, the reasoning engine can determine how to do it using a backward-chaining engine and a knowledge-base.

Another area we are pursuing is dynamic network visualization. Many commercial off the shell (COTS)

network management products can create static network maps, but in a tactical network we need the ability for the maps to change over time as nodes enter and leave. We can use the neighbor data gathered by our protocol to build a network topology map and can update it when nodes leave or join.

REFERENCES

[1] E. Idhaw, J. D'Amelia, J. Burdin, and J. Shaio, "Techniques for enabling dynamic routing on airborne platforms," in *IEEE Military Communications Conference*, 2009, pp. 1 –9.

[2] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace (RFC 4122)," http://www.ietf.org/rfc/rfc4122.txt, [Online; accessed 2011].

[3] A. Adams, J. Nicholas, and W. Siadak, "Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised)," http://www.ietf.org/rfc/rfc3973.txt, [Online; accessed 2011].

[4] "Zero Configuration Networking (Zeroconf)," http://www.zeroconf.org/, [Online; accessed 2011].

[5] "Cisco Discovery Protocol (CDP)," http://www.cisco.com/en/US/tech/tk648/tk362/tk100/tsd_technology_support_sub-protocol_home.html, [Online; accessed 2011].

[6] M. Burgess, "Cfengine: a site configuration engine," in *USENIX Computing systems*, 1995, vol. 8.

[7] W. Enck, T. Moyer, P. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. Greenberg, Y.W.E. Sung, S. Rao, and W. Aiello, "Configuration management at massive scale: system design and experience," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 323–335, 2009.

[8] "NETCONF WG," http://www.ops.ietf.org/netconf/, [Online; accessed 2011].

[9] J. Strassner, *Policy-based network management: solutions for the next generation*, Morgan Kaufmann, 2004.