Secure Remote Management and Software Distribution for Wireless Mesh Networks

Thomas Staub, Daniel Balsiger, Michael Lustenberger and Torsten Braun
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
Switzerland
{staub|balsiger|lustenbe|braun}@iam.unibe.ch

***Abstract*— Wireless mesh networks (WMN) are usually spread over large physical areas. They can include node locations that are difficult to reach, e.g., roof tops. Physical access to certain nodes can even be unfeasible depending on bureaucratic or technical problems. During the life time of a WMN it is necessary to process reconfigurations and software updates. Configuration errors and faulty software updates may then destroy the access to individual nodes. Costly on-site reconfiguration is required. We propose a secure management architecture for WMNs handling configuration errors as well as faulty software updates and avoiding on-site repairs. The architecture is tailored to productive and extensive testbed networks, in which reconfiguration is even more frequent. It is a fully distributed management solution and provides fallback solutions for configuration errors, and kernel panics. The paper presents our architecture and its implementation including the Linux image, the development system and the management console.**

I. Introduction

Wireless mesh networks (WMN) are evolving to an important access technology for broadband services. There are multiple deployments of WMN related to research, e.g. MIT Roofnet [1], [2], Orbit project [3], Microsoft Research [4], [5]. Furthermore, there are multiple cities which are currently deploying metropolitan area networks [6]. All these deployments cover geographically large areas. One can imagine that WMNs are deployed in hostile environments such as forests, deserts, or arctic regions. After deployment not all nodes may be physically accessible or the access may be very complicated and therefore costly.

Reconfiguration and software updates are necessary during the lifetime of any WMN. The reconfiguration and update process is a possible point of failure of the network. The network may be disconnected because of wrong configuration or faulty software updates. The change of radio communication parameters can affect the physical topology of the network as well as cut off nodes from the network. Without an automated reconfiguration, which supports the user in case of defective configuration or errors, physical access to individual nodes may be required.

As experimental research becomes more and more crucial in the design of wireless networks, safe reconfiguration and update of the extend testbed networks are important and time-saving issues. UCSB's ATMA [7] provides a management framework for experimental wireless networks. It is based on an additional WMN deployed beside the experimental network. We think that reconfiguration and updates are essential for both productive and experimental environments. Therefore, we prefer a solution that works the same way in both scenarios.

We provide an architecture that offers secure and safe reconfiguration and update of the WMN without the need of additional infrastructure, e.g. wired or wireless back-haul networks. Our architecture guarantees availability of the network despite of configuration errors and faulty software updates. It further provides the possibility to test configurations that are automatically reverted after a certain amount of time, in case of errors.

The paper is organized as follows. In Section II, our architecture with its basic concepts is presented. The following sections show our implementation. Section III describes our used hardware platform. In Section IV, our embedded Linux image is presented. Section V discusses the configuration and update mechanisms. We conclude with Section VI.

II. Architecture

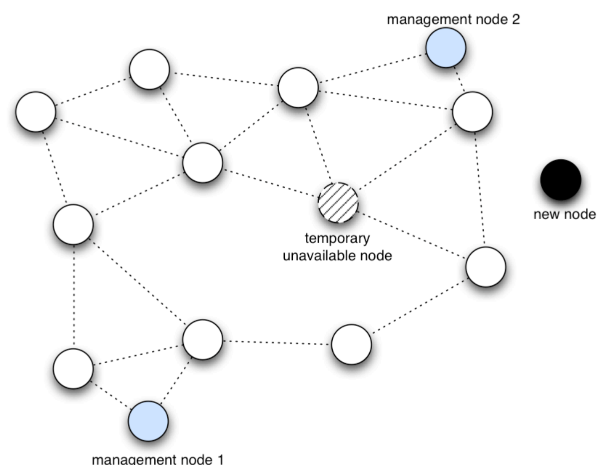


Fig. 1. Example of a WMN: One node is temporally unavailable, e.g., lack of power. Another node is added to the network for the first time. Multiple nodes provide management functionalities for the network.

The target scenario for our architecture is a reconfigurable WMN (see Fig. 1). The WMN consists of multiple wireless

mesh nodes. It is not guaranteed that every node is always reachable. Nodes could be unavailable, e.g. when they have been switched off by users or by loss of power. Attention has been given to these nodes during reconfiguration in our architecture.

For the management of the network either distinct management nodes or ordinary mesh nodes can be used. Management nodes are usually equipped with better hardware than the normal mesh nodes and can provide further features. Monitoring of the network as well as the configuration of all network parameters is the primary task of the management nodes. Their functionalities can be accessed via a web interface. They could further provide tools, e.g., node image generators or a complete development environment.

A. Distribution of Configurations and Software Updates

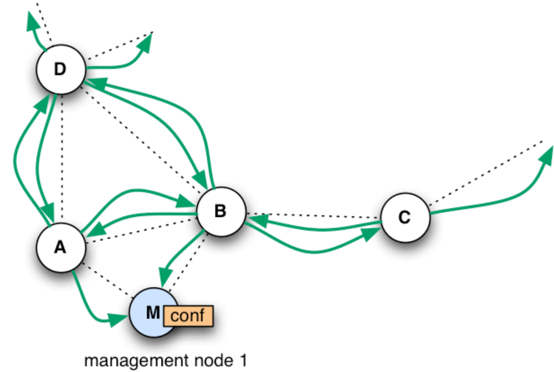
Our architecture disseminates network and node configurations as well as software updates in a distributed way as shown in Fig. 2. Each node is periodically asking its neighbors for newer configurations and software. If updates are available, the node downloads them to its exchange storage. Neighbors of this node will download the updates from there. The downloaded configuration and software updates will be activated after a predefined time.

Nodes that have been down during the distribution of the updates will get the configurations and software updates from their neighbors as soon as they are up again. If critical parameters like wireless communication channel or band have been changed and the awaken node has no connection to any of its former neighbors, it will fall back to its initial configuration and will try to join the network as a brand new node (see Section II-B).

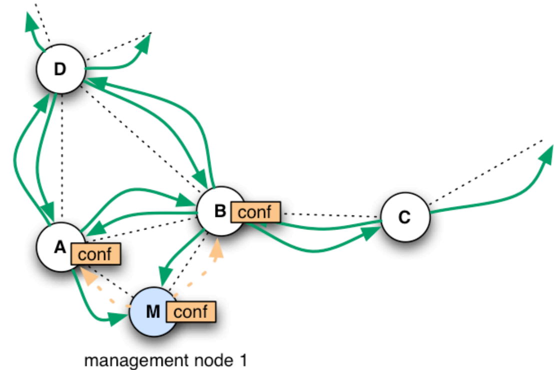
In order to guarantee the connectivity of the network after a reconfiguration, fall back solutions and checks are intended. For example if the transmission power of the wireless radio is reduced, the connectivity of the network is tested. If there is any topology change, the transmission power is stepwise increased until the original connectivity is reached again. Other disruptive changes like wireless channel are also considered in our architecture.

If the user wishes to test a certain configuration, we introduce a temporary update feature in our architecture. The user generates and deploys a test configuration. He further defines a validity time for the new configuration. All nodes backup their configuration, before loading the new one. After the configuration has been fully distributed and set up in the network, a timer on each node is started. The user has now the possibility to check his test configuration. If it satisfies his needs, he can confirm it by sending a confirmation message to each node. The confirmation message stops the timer at the nodes. If the configuration is erroneous or the user did not confirm it, the old configuration will be loaded at the nodes. The network will operate in its last state again.

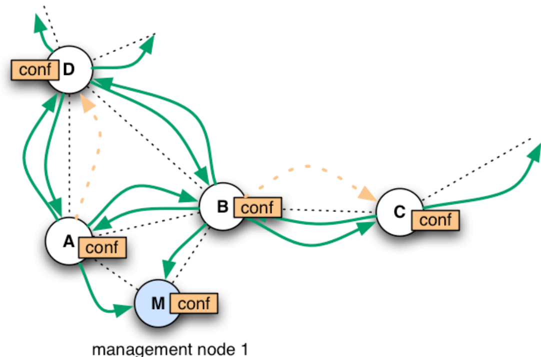
Our architecture provides a safe way to upgrade the node's operating system. The update images are first checked for integrity by the help of hashes and checksums. The updated kernel and filesystem are put in the update storage of the nodes. The system is now instructed to load the operating system only once from the update storage. On the next reboot it would load again from the default storage. If the software update succeeds and the node is up with the new operating system, the update can be made permanent by copying the updates to the default storage. If there occurs any problem while booting the new operating system, e.g., a kernel panic, the system will be automatically rebooted and load the old operating system from the default storage.



(a) Nodes periodically check for updates. A new configuration is injected at a management node (M) or a normal node.



(b) First nodes (A, B) get the update from node M.



(c) Next nodes (C, D) get the update from node A and B.

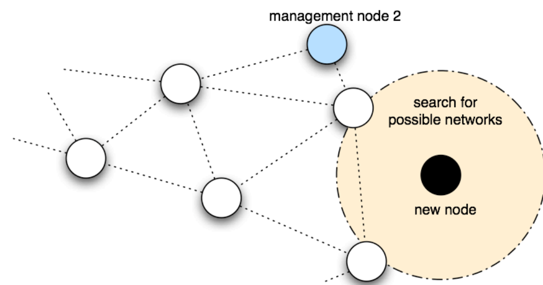
Fig. 2. Distribution of node configuration and software updates.

There exist separated images for configuration of an individual nodes, its state (e.g. its log files), and the operating

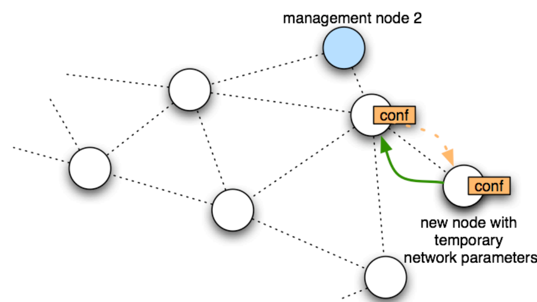
system. This permits the exchange of the operating system without loosing the node's configuration and state. Furthermore, configuration switches do not destroy the state of the node.

B. Integration of a New Node into the Network

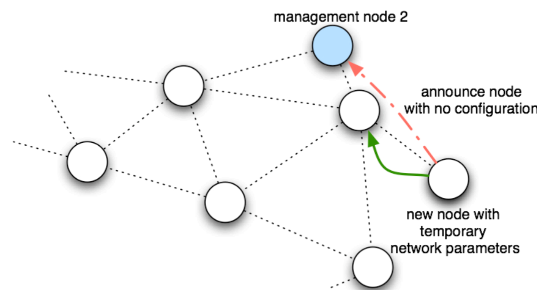
New nodes should be easily integrated into the WMN. Figure 3 depicts the addition of a new non configured node to the network. A standard image has been loaded to the new node. Furthermore, the node has received a unique host name, its public/private key pair as well as the public keys of the other network nodes. The keys are essential to guarantee that only authorized nodes can connect to the network.



(a) New node searches for networks and known peers.



(b) New node sets temporary network parameters and tries to get its configuration from the neighbors. After the new node has received its configuration, it is fully integrated in the network.



(c) If no configuration for the new node exists, the node announces its state to a management node. The user has to generate a new configuration. The new node is integrated in the network after having received the generated configuration.

Fig. 3. Integration of a new node into an existing network.

A new node joins the network by first scanning for active communication channels. On the found channels it searches for IP networks, assigns itself an unused IP address and tries to load configurations from its neighbors. The node authenticates its communication peers with the help of the public keys in its storage. The same is done by the network nodes. They only provide configurations and software updates to known nodes. Therefore, the public key of the new node has to be distributed to all network nodes before the node can join the network. We encourage to use a pool of key pairs when setting up a network. All public keys are then loaded on all nodes at setup time. If there are no key pairs left in the pool for a new node, the additional public key of the new node has to be loaded on all network nodes by the distribution mechanism described in Section II-A. As the image for a new node is usually created at the management node, the distribution of an additional key is invoked automatically if necessary. The configuration of the new node can be already distributed in the network. In this case, the new node simply loads its configuration from one of its neighbors and is then fully integrated in the network. If there is no configuration available, the node signals its lack of configuration to any management node found in the network. The user is then prompted to generate a configuration at the management node.

III. Hardware

For our wireless mesh network we use the Wireless Router Application Platform (WRAP) from PCEngines [8]. Our nodes are WRAP2.C and its RoHS (EU restriction of the use of certain hazardous substances in electrical and electronic equipment) compliant successor board WRAP2.E. It is an embedded board with 233 MHz AMD Geode SC1100 CPU, 128MB RAM, Compact Flash card slot, one Ethernet port, two miniPCI sockets and one serial port. We have preferred WRAP to any Linksys Router based solution with OpenWrt [9] because of its ability to carry two wireless miniPCI cards. This enables multi-radio/multi-channel communication. Our nodes are equipped with two Atheros 802.11a/b/g miniPCI cards. We have further added a 3V Lithium coin cell as battery backup for the real time clock of the node.

IV. Embedded Linux Image for Wireless Mesh Nodes

Existing solutions (like OpenWrt [9]) do not meet all our requirements or are tailored for other hardware than the WRAP platform we use. Our intention is to provide a node image, which is as small as possible while providing maximum functionality. We have achieved this by using special software written for embedded systems. Our selection includes busybox [10] as a replacement of common UNIX utilities and uClibc [11] as small C library. Busybox is a well-known tool for small or embedded devices. It combines tiny versions of many common UNIX utilities (e.g. *ls*, *dmesg*, *top*, *date*) into a single small executable with a size of only 712 KB in our case.

By the help of busybox and uClibc we provide a platform where standard software could be used, e.g., bash, openssh or openssl. This makes the image easily extensible and customizable. We further provide a development system, on which newly required software can be compiled and installed to

the node image. With an existing solution, adding such new functionality can be very difficult. The result is an all-purpose image which looks nearly like a *standard* Linux system. Our image includes the following security features that are also described in Hardened Linux From Scratch (HLFS) book [12]:

- Position Independent Executable (PIE) [13]
- PaX [14]
- Grsecurity [15]
- Stack Smashing Protector (SSP) [16]

A Position-Independent-Executable (PIE) [13] is an executable which is a hybrid of a shared library and a normal executable. Programs compiled as PIE appear as *shared object*. The executable behaves like a shared library. Its base addresses can be relocated. In our image all object code is position independent and the *grsecurity* kernel [15] prohibits text relocation. This closes a security hole that could enable attackers to modify the memory and execute their own code. PaX randomizes the return addresses of PIE programs with Address Space Layout Randomization (ASLR). This further prevents that attackers could take advantage of security bugs as the return addresses are not known to them.

Stack Smashing Protector (SSP) [16] has been developed for protecting applications from stack smashing attacks. This is the largest class of attacks. The protection uses minimal time and space overhead while protecting all functions.

All the described features are used by default when compiling software on the development system.

The resulting image uses about 24.0 MB in uncompressed form in RAM and compressed less than 10 MB on the flash device. Nodes with 128 MB of RAM have still more than 100 MB free for applications.

A. Boot Process

The Compact Flash card has two partitions. Two partitions are needed for safe kernel updates (see Section IV-C). Partition 1 (*/dev/hda1*) contains kernel images, the corresponding root filesystem images, and some boot loader files. Partition 2 (*/dev/hda2*) holds all configuration images, a state directory, and some boot loader files (see Section IV-B).

Normally, the Linux standard boot loader *grub* starts the default image from the first partition (*/dev/hda1*) of the flash device. An image consists of three files: the kernel image itself, a compressed filesystem archive (*.cpio initramfs*) and the *sha1/md5* checksum file. The filesystem archive is loaded into RAM and mounted as root (*/*) at the very beginning of the boot process. The whole system lies therefore in RAM in order to gain performance and to take care of the limited write cycles of a Compact Flash card. Compared to an ordinary RAM disk *Initramfs* requires no fixed size in RAM and can grow and shrink as needed. The whole root tree is writable. As soon as the root filesystem has been mounted, the init process is executed. The init scripts first create device nodes, load the configuration files from the actual configuration image and state files from the state directory (described in detail in Section IV-B). Afterwards, configured services like system logging, web server, secure remote shell (SSH), network time (NTP) and a terminal on the serial line are started. If a network configuration is available at the node, network devices and network parameters are set up accordingly.

B. Individual Configuration and State

As all the files are kept in an *initramfs* archive, they can be changed individually while the system is running. But changes are not saved over a reboot due to the reload of the original archive at the next boot. Therefore, a procedure is required to save files permanently over reboots. Examples are files like */var/log/wtmp* and */var/log/messages*, several individual node and network keys, configuration files for individual node setup, password files etc.

The *config* directory on the second flash partition (*/dev/hda2*) contains different configuration images. A configuration image is an *ext2* loopback image and contains user defined files, which are loaded at boot time by the init scripts as early as possible before any node configuration is done. Each configuration image contains a list of the files kept in it and their destination in the real system. The list is contained in the file */etc/conffiles* on the real system. This file can be adapted in order to add files which must be saved over a reboot.

A node can have more than one configuration image on the flash device (*/dev/hda2*). The */etc/init.d/rc.config* command lists all existing configuration images, the actual configuration image in use, loads or stores configurations from or to configuration images, and creates new configuration images from the actual system configuration.

Some files should not be stored in the configuration image explained above because they should not be replaced in case of configuration switches. For example the file */var/log/lastlog* should be loaded and saved anyway at each reboot independently from a specific configuration image in order to store reliable information on the last logins. All files of this kind represent the state of the node. We store such files in the *state* directory on the second partition. All files in the *state* directory are loaded by the init scripts at boot time and stored when the system reboots. The current log files are saved back to state directory and new empty files are created at each boot. The maximum space that different versions of log files may occupy on disk can be configured. If this limit is reached, the oldest log files are deleted. The */etc/init.d/rc.syslog* script shows all current log files and maximal quota for log files.

C. Safe Kernel Updates

The *grub* boot loader is able to perform the following actions at boot time according to its configuration file *menu.lst*:

1) Install the MBR pointer to another boot partition (this has the effect that the other partition is booted the next time).
2) Boot the operating system from the current boot partition.

These actions provide us the opportunity to boot an update kernel, and let the system fall back to the default kernel when the update kernel fails to boot (e.g. kernel panic). The procedure is depicted in Fig. 4. The following sequences

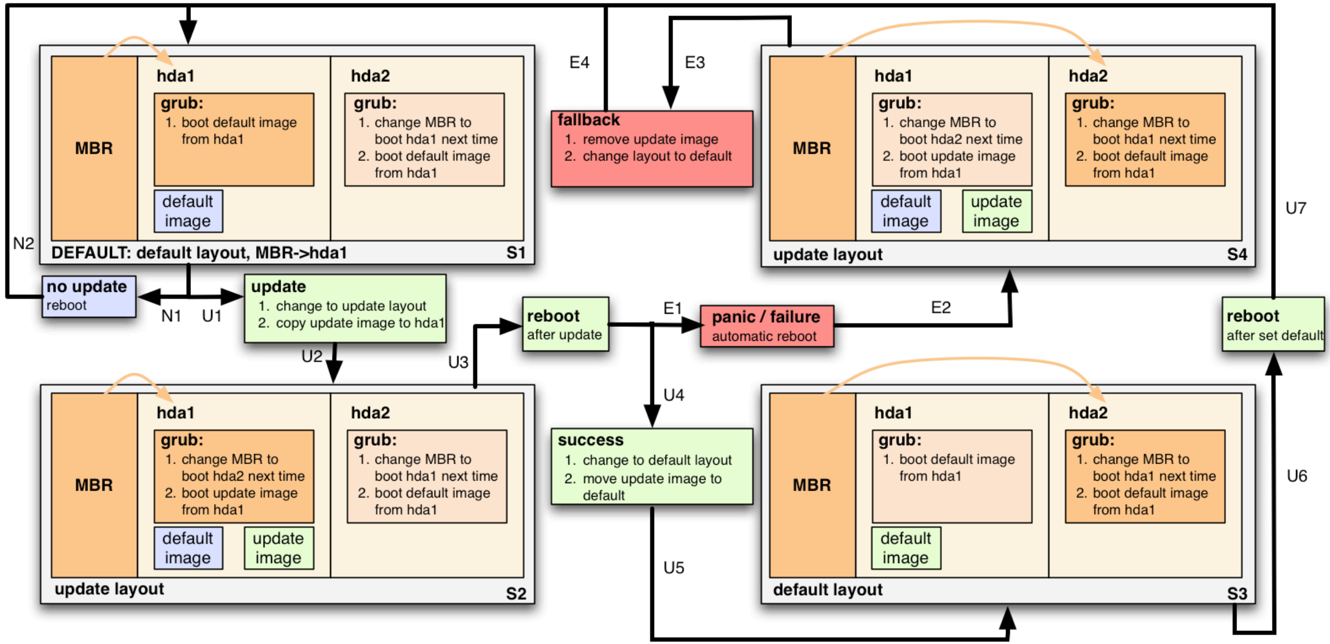


Fig. 4. Safe image and kernel update process with fallback.

provide examples for normal operation, a successful update, and a faulty update:

1) Normal operation: The system is in default configuration (S1). No update is planned. Therefore, the system remains in default configuration after a reboot (N1/N2).

2) Successful update: The system is in default configuration (S1). MBR points to */dev/hda1*. The default image would be loaded after reboot. An update is intended (U1). The layout of *grub* is changed to update layout (U2). The update image is copied to */dev/hda1* (S2). As MBR points to */dev/hda1*, from where *grub* configuration is read at the next reboot (U3). *Grub* sets MBR pointer to */dev/hda2* and loads the update image. If the update has been successful, the layout is reverted to the default layout (U4/U5) and the default image is replaced by the update image (S3). During the next reboot the boot loader (*grub*) configuration on */dev/hda2* is read. MBR is changed to point to */dev/hda1* again. The default image is loaded from */dev/hda1* (U6/U7). The node returns to normal operation (S1).

3) Faulty update: The system is in default configuration. MBR points to */dev/hda1* (S1). The update image is copied to */dev/hda1* and the update layout is set (U1/U2/S2). The system is rebooted (U3). MBR is reset to point to */dev/hda2*. The update image is loaded. The update image produces a kernel panic (E1). The node is automatically rebooted (E2) and is now in error state (S4). As the MBR points to */dev/hda2*, the MBR is reset to boot */dev/hda1* next time and the default image is loaded (E3). The node runs with the old kernel again. The update image is removed, the layout is reset to default (E4). The node returns to normal operation (E4/S1).

The update of each node concerns only the kernel and the corresponding *initramfs* image, which contains all basic software of the system (configuration and state files are treated separately as shown in Section IV-B). In order to manage the configurations there exists the */etc/init.d/rc.update* script, which can initialize updates, detect working and failed updates and make updates permanent.

The script includes consistency checks of the included files. It checks the compressed kernel and the *initramfs* images by comparing the *sha1/md5* checksums, and the *grub* configuration file *menu.lst* by parsing the file and checking the content to the newly calculated form.

V. Configuration and Management Software

A. Management Console

For network management, we provide a LiveCD for the Linux management node. It serves as the starting point for any configuration. If the LiveCD detects an USB-storage device at boot time, it loads its configuration (ssl-certificates, passwords, node-definitions, cfengine-keys). If no USB device is detected with these configuration files, the LiveCD prompts the user to provide the initial configuration parameters on console. Once the LiveCD has its initial configuration, the user connects with a web browser to the LiveCD's SSL web server. The new network can be defined on the setup page. One has to provide the number of nodes, their host names and some passwords. Then the LiveCD generates individual images for each node. The user installs each image on the corresponding node.

On the web interface each node's network setup may be configured. This can either be done before generating the images or afterwards when the nodes are already deployed. In the second case the nodes will get their configuration from one of their neighbor nodes if their configuration is already

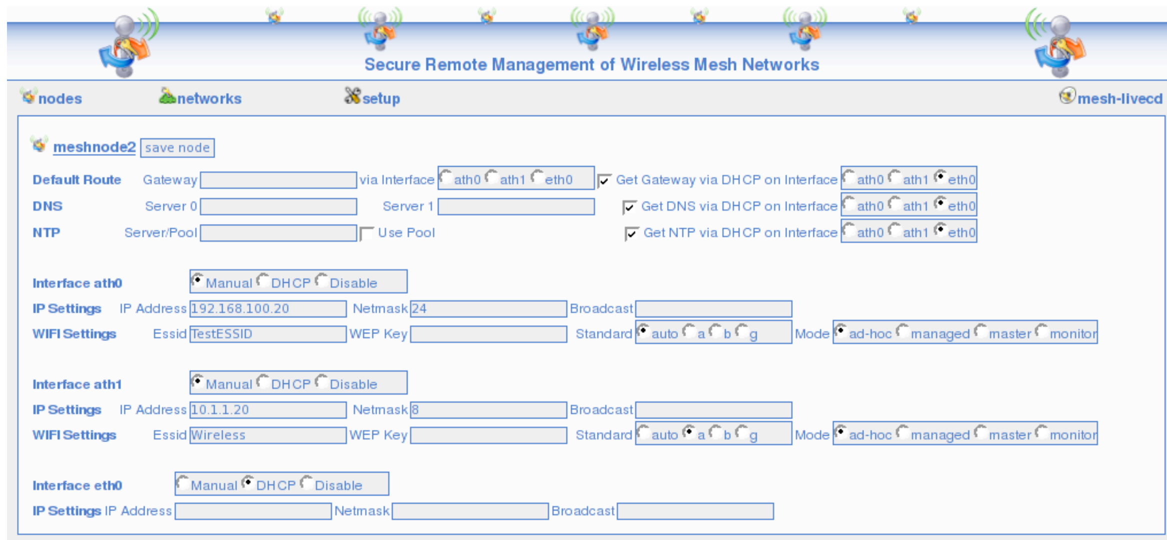


Fig. 5. Management console: individual configuration page for one node.

available in the network or else signal the management console that they have no configuration. Figure 5 shows a screenshot of the web interface.

The user can adapt the configuration of the network at any time. After a configuration has been setup, the new configuration has to be committed in order to be distributed by *cfengine* [17]. The whole configuration of the network, including the current configuration of the LiveCD itself can be stored on an USB-storage device, and restored at the next start of the LiveCD. Therefore one LiveCD can be used for managing more than one network. Its minimal hardware requirements are: i586 compatible processor, 128 MB RAM, Atapi/IDE cdrom device, USB port.

The LiveCD offers a development mode besides the management console mode. Started in development mode, the LiveCD acts as full development system for the node image. This functionality requires a free hard disk partition. In development mode, the user is able to compile and install new needed software for the node-image.

B. Distribution of Configuration Parameters

For the distribution of all configuration and possible updates we use *cfengine* [17], [18], a powerful utility for organizing and distributing system administration tasks in a network.

We use a distributed design as presented in Section II-A. The only static parameters are the host names and the unique public/private key-pair for each node. The configuration of the network is dynamic and can be done at any time. In order to work in such an environment *cfengine* is configured to authenticate by host name and security key pair.

As the hosts have to be able to look up their names to the current IP addresses even if no external name service is available, the nodes have an individual */etc/hosts* file. Each time *cfengine* is executed, the */etc/hosts* is dynamically created. The script *netcfpeers.sh* tests the node's peers with *traceroute*, writes the */etc/hosts* file and returns a list of available peers to *cfengine*. Therefore, *cfengine* is able to distribute all of the settings over different IP networks even if they are dynamic. Each node stores all configurations of the networks. The public keys of all nodes are distributed. This guarantees that every node knows its neighbors.

Cfengine offers a lot of flexibility by its concept of dynamic grouping of nodes into classes. The membership of a node to a certain class is dynamically set by execution of any script. According to the class all other actions of *cfengine* are defined. We take advantage of this flexibility in our concept by defining appropriate classes and actions.

The architecture of *cfengine* is based on pulling the desired information from its peers. But it is possible to simulate a push method by invoking the pull mechanism remotely. We do not use this functionality and only rely on pulling. Every node is server and client at the same time. In order to serve requests for updating configuration files, the *cfservd* daemon is running on each node. It only grants access to known peers. Further, all transmissions of *cfengine* are encrypted. The pulling mechanism *cfexecd* is executed by *crond* every two minutes in our current setup, but the frequency can be easily adapted. *Cfagent* first tries to update the configuration of *cfengine* from its calculated peers within a random time-offset of up to one minute. This reduces the probability of too many simultaneous connections. After updating, *cfengine* carries out administrative tasks. The current state of the node is checked by scripts in order to classify the node for a particular class. Afterwards, the new configuration is copied by comparing the modification time of each file. During each run *cfengine* tries to gather new information about the network from its peers by copying the *network.test* directory. Periodically, every 15 minutes, *cfengine* checks for other updates like changed configuration parameters or system updates. For example, a class is defined in *cfagent.conf* for updates. A node is put

in this class if it receives a positive exit value when *update-test* script is executed. If *cfagent* finds itself executed on a node that has a newer version of these files available it will just interact with the interface described in Section IV-C to perform the update.

C. Network Update with User Interaction

If a new network configuration for a certain node is desired, the user creates the configuration, e.g., with the management console. The new configuration file is copied by the management console (or manually) to the *exchangefiles* directory. Further, the user defines the wait cycles (intervall between two *cfagent* runs, in our case two minutes) until the configuration is permanent.

During the first *cfengine* cycle every node that has the node with the new configuration as its own peer, gets the information about the new configuration. The configuration is not further processed, except of publishing it to other nodes.

The *cfagent* realizes the newer configuration files in the *network.test* directory. The node is classified to belong to a new class. This invokes an external bash module that takes care of the setup. It discovers whether any dynamical network setup needs to be done (DHCP). If this is the case, the node delays the update to the next cycle. This procedure is repeated until there is only statical configuration information. We have configured *udhcpc* to virtually change the state of the device from dynamic to static after having received the IP address from the DHCP server. This static configuration is written back and propagated as the new configuration to the node's peers. After all nodes have static IP addresses, the individual nodes save their current configuration and remove the *user interaction* file from previous updates. Further, they read the number of wait cycles to keep the new configuration before falling back to the old configuration. Then each individual node calculates the new */etc/hosts* file and the changed interfaces (and only those) are restarted on the reconfigured node. The described update procedure does not happen simultaneously, but is done in a completely de-central way.

After the update each node indicates its readiness by touching a file in its *exchangefiles* directory. As soon as multiple nodes are up again, the update notifications are distributed over the new evolving network. The nodes are now waiting for a user interaction during the defined fallback period. If no user interaction has taken place, the nodes copy back their old configuration and restart the affected network devices.

A user can check the state of the network on every single node or over the web interface. If the network satisfies the user's requirements, he confirms the network to keep the current configuration. The confirmation message has to reach all of the nodes before they counted down their own wait cycles (timer). If confirmed, the nodes set the current configuration to default, disable the timer, and remove the old configuration.

If the network is in inconsistent state after partial successful updates, it is recommend to define a timeout, after which a node that has no connection to its previous neighbors reloads the initial configuration and tries to join the network as a new node (see Section V-D).

D. Plug&Play Integration of New Nodes

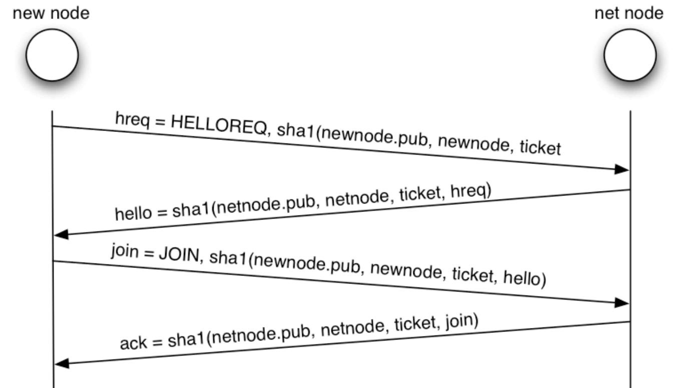


Fig. 6. A new node and its communication peer resolve the *host name to IP address* mapping in order to exchange configuration.

A new node N_{new} can join the deployed network and is automatically configured, if it has a working base image with the necessary keys. Image and keys can be generated by the management console. There is no configuration needed at the creation time of the image. All parameters including the network configuration can be set, when a node joins the network. There are two situations, in which a node is treated as a new node:

1) The node had no connection for a given period of time and thus falls back in new node mode. The node has already all necessary keys of the network.
2) A brand-new node does not have all public keys of the network. Either the node is created by the management console and receives the public keys of the network as well as its own public/private key pair with its image, or the administrator has to copy all existing keys to the new node manually. In both cases, the public key of the new node has to be distributed to all existing nodes. The management console takes care of all this work and will distribute the keys using *cfengine*.

N_{new} searches in all predefined configuration networks if anyone is reachable (overhearing or active checks). If an active node is found, the node selects an unoccupied IP address in its IP network and tries to make two specific https requests. The search is repeated until N_{new} receives the correct answers on its https requests. A more detailed view on the requests is shown in Fig. 6:

1) N_{new} connects to *https://netnode/newnode.cgi* and transmits *hreq= HELLOREQ,sha1(newnode.pub, newnode, ticket)* with *ticket=floor(SystemTime / TicketValidTime)*. The node in the network N_{net} calculates hashes for each node he knows according the rule: *testhash= sha1(node.pub, node, ticket)*. If one hash matches the received one from N_{new} and if no ticket from this node

was received the last *TicketValidTime* seconds, N_{net} knows the name and the IP address of N_{new} and returns *hello=sha1(netnode.pub, netnode, ticket, hreq)*.

2) N_{new} checks with the same procedure as N_{net} in step 1 if the received *hello* matches any known node. N_{new} sends the message *join=JOIN,sha1(newnode.pub, newnode, ticket, hello)* via *https://netnode/newnode.cgi* to N_{net} to acknowledge the received message. N_{net} recognizes the *join* message as it knows the *hello* hash and can calculate the *join* hash. N_{net} now writes IP and host name of N_{new} to its */etc/hosts* and replies with *ack=sha1(netnode.pub, netnode, ticket, join)*.
3) N_{new} checks *ack* and writes N_{net} host name and IP address in its */etc/hosts*.
4) Configuration of N_{new} can now done by *cfengine*. If a configuration for N_{new} is already distributed, N_{new} will get it by *cfengine*, otherwise N_{new} will show up as a node waiting for configuration in the management console.

There are some limitations of the described procedure. As it would take a long time to search every possible IP network, it is recommended to define some configuration networks. For security reason (reply attacks) a new node can join a specific node in the network only once in *TicketValidTime* seconds. Therefore, if messages are lost, the node has to wait until the ticket is invalid before its next try to join the same node in the network.

VI. Conclusion and Future Work

We have presented a distributed secure and safe management architecture for WMNs. It supports the user in the configuration task, and guarantees network availability even after configuration errors or updates with faulty software images. It does not require any additional infrastructure. The whole configuration is done in-band. It offers timed updates. A configuration can be tested and in case of errors the node reverts to the old configuration after a certain amount of time.

As part of future work, we have planned extensive testing of the described solution and support for IPv6. IPv6 would provide unique IP addresses for all nodes. It simplifies dynamic setup, mobility management as well as security in a WMN. As most parts of the embedded Linux already support IPv6, only extensions to some configuration scripts are needed. IPv4 Zeroconf protocols (e.g. multicast DNS) will be integrated in our next release. We further focus on extensions of configuration interface to include gateways to wireless sensor networks. Other open issues are modular enhancements of the management console in order to provide easy integration of new configuration options, e.g. additional routing protocols, experimentation setups.

Acknowledgement

The work presented in this paper was partly supported by the Swiss National Science Foundation under grant number 200020-113677/1.

References

[1] J. Bicket, D. Aguayo, S. Biswas, and R. Morris, "Architecture and evaluation of an unplanned 802.11b mesh network," in *MobiCom '05: Proceedings of the 11th annual international conference on Mobile computing and networking*. Cologne, Germany: ACM Press, August 2005, pp. 31–42.

[2] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris, "Link-level measurements from an 802.11b mesh network," in *International Conferences on Broadband Networks (BroadNets)*, 2004.

[3] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh, "Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols," in *WCNC 2005 IEEE Wireless Communications and Networking Conference*, vol. 3, March 2005, pp. 1664 – 1669.

[4] R. Draves, J. Padhye, and B. Zill, "Routing in multi-radio, multi-hop wireless mesh networks," in *10th annual international conference on Mobile computing and networking MobiCom '04*. Philadelphia, PA, USA: ACM Press, 2004, pp. 114–128.

[5] ——, "Comparison of routing metrics for static multi-hop wireless networks," in *Conference on Applications, technologies, architectures, and protocols for computer communications SIGCOMM '04*. Portland, Oregon, USA: ACM Press, August 2004, pp. 133–144.

[6] R. Karrer, A. Sabharwal, and E. Knightly, "Enabling large-scale wireless broadband: The case for taps." in *2nd Workshop on Hot Topics in Networks (Hot-Nets II*, Cambridge, MA, November 2003.

[7] C. C. Ho, K. N. Ramachandran, K. C. Almeroth, and E. M. Belding-Royer, "A scalable framework for wireless network monitoring," in *2nd ACM international workshop on Wireless mobile applications and services on WLAN hotspots WMASH '04*. New York, NY, USA: ACM Press, 2004, pp. 93–101.

[8] PC Engines GmbH, "Wireless Router Application Platform (WRAP)," www.pcengines.ch, 2006. [Online]. Available: www.pcengines.ch

[9] M. Baker, G. Rozema, I. Kaloz, N. Thill, F. Fainelli, F. Fietkau, M. Albon, and T. Yardley, "OpenWrt," http://openwrt.org/, 2006.

[10] R. Landley, "Busybox," http://www.busybox.net, 2006.

[11] E. Andersen, "uclibc," http://www.ulibc.org, 2006.

[12] HLFS Development Team, "Hardened Linux From Scratch (HLFS)," http://www.linuxfromscratch.org/hlfs, 2006.

[13] J. Jelinek, "Position Independent Executable (PIE)," http://gcc.gnu.org/ml/gcc-patches/2003-06/msg00140.html, June 2003.

[14] PaX Project, "Pax," http://pax.grsecurity.net/, 2006.

[15] B. Spengler, "Grsecurity," http://www.grsecurity.net/, 2006.

[16] H. Etoh, "Stack Smashing Protector (SSP)," http://www.trl.ibm.com/projects/security/ssp/, August 2005.

[17] M. Burgess, "Cfengine: a system configuration engine," http://www.cfengine.org, 1993.

[18] ——, "A tiny overview of cfengine: Convergent maintenance agent," in *1st International Workshop on Multi-Agent and Robotic Systems MARS/ICINCO*, Barcelona, Spain, September 2005.