Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Tux meets Radar O'Reilly—Linux in military telecom

Grant Likely
Secret Lab
grant.likely@secretlab.ca

Shawn Bienert
General Dynamics Canada
shawn.bienert@gdcanada.com

Abstract

Military telecom systems have evolved from simple two-way radios to sophisticated LAN/WAN systems supporting voice, video and data traffic. Commercial technologies are being used to build these networks, but regular off the shelf equipment usually isn't able to survive the harsh environments of military field deployments.

This paper discusses the use of Linux in General Dynamics' vehicle mounted MESHnet communication system. We will discuss how Linux is used in the system to build ad-hoc networks and provide reliability for the soldiers who depend on it.

1 Introduction

To say that good communication is critical to the military is never an understatement. Wars have been won and lost on the basis of who had the most accurate and timely information. It is understandable then that the military takes its telecom gear very seriously.

Just like the private sector, in order to survive, the military has had to keep up with advances in technology. Current military communications networks carrying both voice and data traffic bear little resemblance to the original analog radio systems of the past.

In most cases, however, regular commercial equipment is not suitable for a military environment. Aside from the obvious fact that most equipment isn't painted green, the military puts high reliability expectations on its equipment. When your life depends on the proper operation of your equipment reliability tends to be an important concern. For example, tank drivers typically do not have a very good field of view from where they are located in the vehicle, and thus are dependent on directions provided by the commander via the intercom system to accurately direct the vehicle. An intercom failure can very quickly result in problems like running over a small car or driving through a building.

There is a natural conflict between reliability and the increasingly complex services required by military users. As complexity increases, so do the number of potential failure points which tends to reduce the stability and reliability of the entire system. As new technologies such as WiFi, VoIP and Ad-Hoc mobile networking are added, system engineers need to analyze the impact on the rest of the system to ensure overall reliability is maintained.

Within this environment, Linux and other Free and Open Source Software (FOSS) components are being used as building blocks for system design. As we discuss in this paper, FOSS is proving to be a useful tool for solving the conflicting requirements inherent in large military telecom system designs.

2 Typical Military Telecom Network

For illustration purposes, Figure 1 is an example of a typical military telecom deployment based on the next generation of General Dynamics Canada's MESHnet platform. MESHnet equipment provides managed network infrastructure for voice and data traffic within and between military vehicles and provides voice services which run on top of it. For example, a MESHnet user in a vehicle has an audio headset and control panel that give him intercom to his other crew mates, direct telephone service to other users, and access to two-way radio channels. He also has an Ethernet port for attaching his laptop or PDA to the network for configuration, status monitoring, email, and other services. In the background, GPS and other sensors attached to each vehicle use the radio service to automatically transmit position and status reports back to headquarters [1].

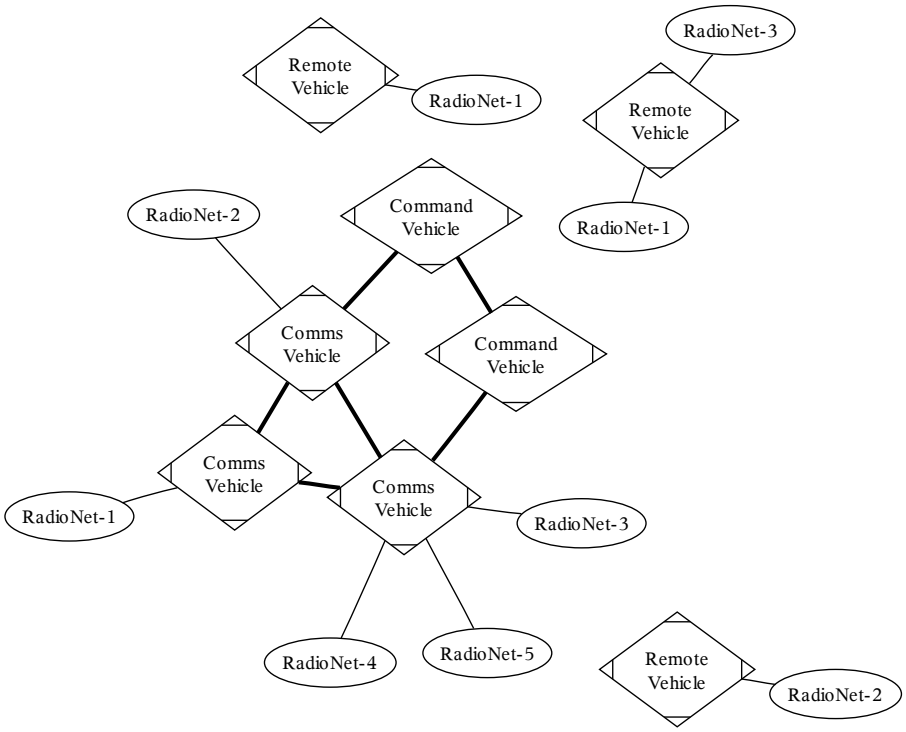

Figure 1: Example MESHnet Deployment with Headquarters and Remote Vehicles

2.1 Vehicle Platforms

At the heart of the MESHnet system are the vehicle installations. MESHnet equipped vehicles provide a user terminal and headset for each crew member in the vehicle, a set of interfaces to two-way radios, and a network router for wired and wireless connections to external equipment. The whole vehicle is wired together with an Ethernet LAN. Figure 2 shows an example vehicle harness which includes 2 radios and a GPS sensor. You'll notice that the Ethernet topology (shown by the bold lines) is a ring. The ring is for redundancy in the event of a cable or equipment failure.

The system is designed to immediately provide intercom service between crew members when the system is initially powered on. Each radio is detected and configured as an abstract *radio net*. Solders can select from a variety of radio nets for monitoring with a single key press and use their Push-to-Talk (PTT) key for transmitting.

Data equipment like Laptops, PDAs, and sensors can be connected to the system via Ethernet, USB, and serial ports, allowing the equipment to work together and providing access to digital data radios for low bandwidth network traffic back to headquarters.

2.2 Headquarters

When a battlefield headquarters is established, several vehicles are parked together and Ethernet is again used to connect them in a mesh topology.[1] The on-vehicle router keeps internal and external Ethernet segments separate. Services provided by each vehicle are bridged onto the inter-vehicle LAN so that user terminals have access to all radios and other equipment within the headquarters.

3 System Design Issues (and how Open Source can solve them)

In this section we discuss the various aspects of the communication system that need to be addressed by system

[1] Hence the name MESHnet.

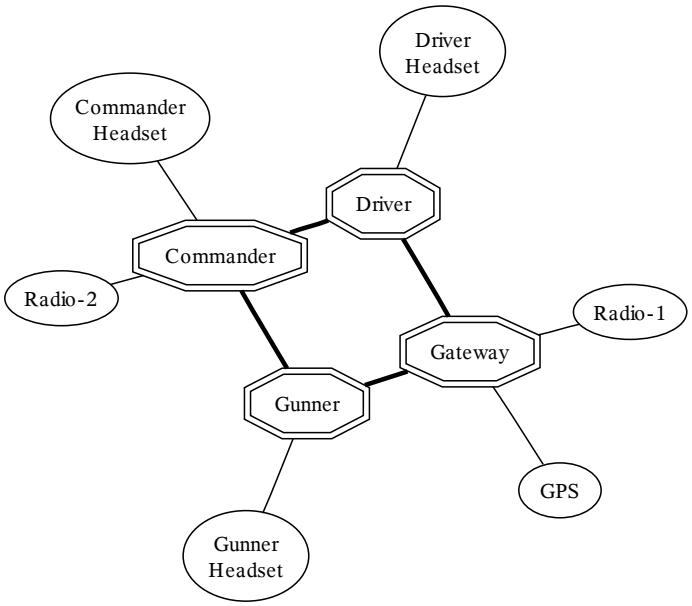

Figure 2: Example MESHnet Equipped Vehicle Installation

designers. Most of these issues bear close resemblance to issues also faced in the commercial environment, but there are some unique characteristics.

3.1 Open Architecture

Telecom systems have become so complex and varied over the years that it is no longer feasible for a single vendor to be capable of designing and supplying all equipment used in a tactical network. Many different vendors supply equipment which is expected to interoperate with the rest of the network. Military customers are also wary of solutions which lock them into a single vendors solution or which force the replacement of existing equipment. As such, there is significant pressure by military customers to use common interfaces wherever possible.

It is now taken for granted that Ethernet and 802.11 wireless are the common interconnect interfaces. The historical proprietary interfaces used in military networks are rapidly giving way to established commercial standards. Not only does it make interoperability easier, it also allows military equipment to use readily available commercial components which in turn reduces cost and complexity.

Additionally, the protocols used by tactical applications are rapidly moving toward common protocols published by international standardization bodies like the Request for Comments (RFC) documents published by the Internet Engineering Task Force (IETF). The entire network is based on the Internet Protocol (IP). Dynamic Host Configuration Protocol (DHCP), Domain Name System (DNS) and the Zeroconf protocol suite simplify and automate network configuration. SNMP is used for network management. Session Initiated Protocol (SIP) and Real Time Protocol (RTP) are natural choices for audio services like radio net access, intercom, and of course, telephony. Similarly, HTTP, SSL, SMB, and other common protocols are not just preferred, but demanded by military customers.

Just like in the commercial world there is little economic sense in developing all the software infrastructure to support these protocols from scratch when the same functionality is easily obtainable from third parties. In particular, since development effort on FOSS operating systems is measured in thousands of man years [2], the functionality and quality FOSS offers is far broader than any company can hope to develop over the course of a single product development cycle.

Of course, custom engineering is still required when standard protocols are not suitable for a particular application. Even so, it is still preferable to start with an existing protocol and build the extra functionality on top of it; ideally while maintaining compliance with the original protocol.

This is where the power of open architecture comes into play. If a custom protocol is what is required, then rather than creating an entirely new protocol from scratch, an existing, standard protocol can be easily tweaked to accommodate the designer's needs. Or, better yet, multiple standard protocols can be combined into a single, custom protocol which still maintains backward compatibility to each original protocol. For example, communicating over a remote radio net involves several tasks: knowing that the radio exists and understanding the path to get there, initiating a pressel-arbitrated session with that radio, and finally transmitting digital audio over the network to that radio. Using a combination of such standard protocols like Zeroconf, SIP, and RTP as building blocks, a custom protocol can be created which will allow the user to perform this complicated operation in a single, simple step with little additional engineering required.

3.2 Segmentation of Functionality

To manage complexity and keep system designers from driving themselves insane with their own designs, it is important to establish boundaries between areas of functionality. Once again, this is not much different from the issues faced by commercial system designers, but the high reliability requirement brings the issue to the forefront.

In the military environment, equipment failures are not just planned for, they are expected. It is important that when equipment does fail that the impact is minimal. For example, imagine a network with 3 user terminals named *commander*, *driver*, and *gunner* and connected in a ring. If the *gunner* terminal fails, then it is expected that any headsets or radios directly connected to the *gunner* node would no longer work. However, that same failure should not affect intercom between the *commander* and *driver* nodes. In this case the system is designed so that no service depends on a particular unit of hardware that is otherwise unrelated to the service. So, while the intercom between *commander* and *driver* may pass through the *gunner* node, the system is designed to bypass it in the event of a failure.

In this example, several boundaries in functionality work together to ensure the desired behaviour. First, at the physical layer the units are at the very least connected in a ring topology which insures that any single point of failure will still retain a connection path between the remaining nodes. A Rapid Spanning Tree Protocol (RSTP) agent runs on each node and controls the network fabric to eliminate Ethernet connectivity loops in real time and ensures the layer 2 network environment is usable without any need for manual intervention.

At layer 3, Zeroconf is used to provide a fallback mechanism for assigning IP addresses in the absence of a properly configured DHCP server. Any two nodes are able to establish a network connection in the absence of any other hardware.

Similarly, the radio net service is logically separate from the network layer and any other services in the network. The radio server runs as a user space application on the node to which a radio is attached and accepts connections from radio clients across the network. It also advertises itself via Multicast DNS (mDNS) which isolates it from depending on a properly configured DNS server. In turn, the radio clients are also user space processes running on end user nodes. If the hardware hosting the radio server fails, then only access to that particular radio is affected and will not bring down the rest of the comm system.

Services and protocols that are decentralized, or are designed to provide automatic failover are greatly preferred over services which do not. The intercom service is provided by *intercom agents* running on each end user devices. The intercom agents do not depend on a single server node to mix all audio streams, but instead use mDNS to discover each other and determine how to mix the intercom traffic between them. Any one intercom node failure shall not take down the entire intercom service.

The Linux kernel plays a significant role here too as the system is designed to tolerate software failures also. With several applications executing on the same hardware device, the failure of a single application must not cause other unrelated applications to fail. The enforced process separation provided by the Linux kernel pre-

vents a bug in one application from affecting the environment of another. This feature in particular is a benefit over traditional flat memory model RTOS systems when providing complex services within a single hardware platform.[2]

Finally, it can be argued that open source projects have tended towards well defined boundaries between components. There is the obvious boundary between kernel and user space, but there are also well defined boundaries between the libraries composing the protocol and application stacks. For example, the MESHnet user devices make use of the GStreamer framework for processing audio streams. The GStreamer core library defines a strict API for plugins but implements very little functionality inside the core library itself. The plugins use the API to communicate with each other over a well defined interface which isolates the two sides from internal implementation details. Separation at this level doesn't directly improve reliability and fault tolerance, but a well defined and predictable API with few side effects makes the system design easier to comprehend and therefore easier to audit for bugs.

3.3 Capability and Quality

Military customers are also unapologetic in their appetite for rich features in their equipment. They want all the features available in comparable commercial equipment, without sacrificing the reliability requirements discussed above. As already stated, the budgets of most military equipment development projects are not large enough to fund the development of all the required components from scratch, so system designers must look to third party software to use as the starting point. This means either licensing a proprietary application or selecting a FOSS component.

When evaluating third party components, quite a few questions tend to be asked: How well does it work? Does it support all the features we need? Is it under active development? Do we get access to the source code? Can it be customized? How much does it cost? How much work is required to integrate it into the rest of the project?

FOSS components do not always come out on top in the tradeoff analysis. For some of the questions, FOSS has a natural advantage over its proprietary counterparts; access to source code and favorable licencing terms being the most significant. For others a lot depends on the type of application and what FOSS components exist in that sphere.

For example, within the operating environment sphere there are the Linux and BSD kernels, several other open source RTOSes, and various commercial offerings like vxWorks and QNX. In most cases, Linux comes out on top; active development is high which inspires confidence that the kernel will be around for a long time. It boasts a large feature set, the code quality is excellent, and is easy to port to new embedded platforms.

However, there are still areas where Linux is not chosen as the solutions for good reason. Systems with tight memory constraints are still the domain of traditional RTOSes or even bare metal implementations. Despite large strides being made in the area of Linux real time, some applications require guarantees provided by traditional RTOSes. The existence of legacy code for a particular environment is also a significant factor.

Emotional and legal influences also have an impact on what decision is made. Emotions come into play when designers already have a bias either for or against a particular solution. Misunderstanding or mistrust of the open source development model can also dissuade designers from selecting a FOSS component. And there is always ongoing debate over legal implications of using a GPL licensed work as part of an embedded product.

No single aspect can be pinpointed as the most important factor in selecting a component for use. However, if a FOSS component does provide the functionality needed, and it is shown to be both reliable and actively used in other applications, then there is a greater chance that it will be selected. Well established projects with a broad and active developer base tend to have an advantage in this regard. Not only does this typically indicate that development will continue over the long term, it also suggests (but doesn't guarantee) that it will be possible to obtain support for the component as the need arises. It is also often assumed that a large existing user base will contribute to code quality in the form of bug fixes and real world testing.

Smaller and less popular FOSS projects are at a bit of a disadvantage in this regard and can be viewed as a bit of a risk. If the project developers stop working on it

[2] Granted with the tradeoff that embedded Linux system are typically larger and more resource-hungry than the equivalent RTOS implementation. There are no claims of something for nothing here.

for one reason or another, then the system vendor may be forced to assume the full burden of supporting the software in-house. That being said, the risk associated with small FOSS components is still often lower than the risk associated with a proprietary component being dropped with no recourse by its vendor.

3.4 Maintenance

Unlike the consumer electronics market with high volumes and short product life cycles, the military equipment market tends towards low volume production runs and equipment which must be supported for decades after being deployed. Equipment vendors, especially of large integrated systems, often also enter into long term support contracts for the equipment they have supplied.

Knowing this, it is prudent to start the design process in the mindset that any chosen components such as CPUs and memory chips must be replaceable for the entire expected lifetime of the equipment. One way to do this is to restrict component choices to ones that have a long term production commitment from the manufacturer. Another way is to do a lifetime buyout for the quantity of chips required over the expected support period.

Similarly, software components have the same support requirement. Each component must be supportable over the long term. FOSS components have a natural advantage in this area. Unlike with proprietary components, a third party FOSS vendor cannot impose restrictions on the use and maintenance of a FOSS component. Nor can business or financial changes with a third party vendor affect the ability to maintain the product.[3]

3.5 Redundancy and Fault Tolerance

In this section and the next we get into the real areas where military equipment vendors differentiate themselves from their competition. Pretty much all of the *functionality* required for tactical telecom systems already exist to a large degree in both the FOSS and proprietary ecosystems. How well the components are *integrated* together onto a hardware platform is a big part of whether or not the system is suitable for military use, and that depends on strong system engineering.

[3]Unless, of course, you've also subcontracted support to said third party vendor, then you could be stuck with a manpower problem.

Fault tolerance is an excellent example. Looking at individual components does not tell you much about the system as a whole. To design a reliable system requires looking at the entire system requirements and designing architectures that provide those functions in a reliable and fault tolerant way. Some of those decisions are simple and only affect a small aspect of the design. For example, a typical requirement is for equipment to stand up to the kind of abuse inflicted by solders. A common solution is to enclose the electronics in cast aluminum chassis and to use MIL-STD-38999 connectors instead of RJ-45 jacks for cable connections.

Other issues affect the design of more than one subsystem. Designing a reliable Ethernet layer has an impact on multiple layers of the system design. It has already been discussed that using a mesh topology of Ethernet connections requires the design of each node to include a *managed* Ethernet switch and requires an RSTP agent to run on each box. In addition, the system must be able to report any relevant changes to the network topology to the system users in a useful form. For example, on a vehicle with three nodes connected in a ring, if the system detects that one of the three links is not connected, then that probably indicates that an equipment failure has already occurred and that there is no remaining backup connections in the event of a second failure. This is information that the soldier needs to know so that a decision can be made about whether or not to continue using the damaged equipment. Therefore, the reliability of the Ethernet layer also has an impact on the design of the user interface so that changes in equipment status can be reported.

While individual components have little influence on the system design as a whole, using well designed components with predictable behaviour simplifies the job of the system designer just by requiring less effort to understand the low level intricacies.

3.6 User Interface and Configuration

Finally, even the most feature rich and capable a system is just an expensive doorstop if nobody is able to understand how to use it. On the whole, solders are mostly uninterested in the details of a telecom system and only care about whether or not the system lets them talk to who they need to talk to and provide the network connections they need. Even if the system is quite complex

the system designer should strive to make it have the appearance of simplicity for the vast majority of users.

For example, if the network consists of user devices with attached headsets and two units are wired together and powered up without any kind of network configuration, then it is appropriate for the devices to self configure themselves and enable intercom between the two headsets without any manual intervention. Similarly, selecting basic services should strive to only require a single key press. For the few users who do need greater control, like network managers, it is appropriate to provide a different control interface that doesn't hide the details or complexity of the system.

4 Conclusion

The military sector faces significant challenges when designing a communication system that meets the demand for increasingly complex functionality while still retaining the robustness and reliability required by life-critical equipment. With its open architecture and high level of functionality, quality, availability, and maintainability, Linux and other Free and Open Source Software is often well suited to providing the building blocks on which to base the next generation of sophisticated yet stable and operationally simple military communication systems.

References

[1] Mark Adcock, Russell Heal, A Land Tactical Internet Architecture for Battlespace Communications, `http://www.gdcanada.com/documents/Battlespace%20Networking%20MA%203.pdf`, Retrieved on Apr 10, 2008

[2] David A. Wheeler, More Than a Gigabuck: Estimating GNU/Linux's Size, `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`, Retrieved on Apr 10, 2008