Code Propagation in Wireless Sensor Networks

Sjoerd Langkemper

April 30, 2007

Abstract

Wireless sensor networks consist of many radio-equipped nodes which are limited in resources and power. After deployment, these nodes may need reprogramming to fix bugs, add features or make the nodes suitable for another task, among others. Because it may be impractical to physically visit these nodes to reprogram them, several methods have been proposed to reprogram nodes using data sent over the network. This paper gives an overview of methods to propagate code through the network and to reduce the size of this code.

1 Introduction

In a wireless sensor network (WSN), nodes can communicate with each other through radio. Since the radio has a limited range, not all nodes may have a direct link to any other node. Furthermore, nodes are often battery-operated and therefore have limited power. WSNs can be used to monitor for example animal habitat [Lev02], enemy territory, water contamination and seismic activity.

Nodes are typically programmed before deployment. However, after deployment, it may be useful to reprogram the nodes. Reprogramming may be helpful for re-tasking a deployed network, fixing bugs, introducing new features and tuning the system parameters to the operating environment. [Bal06]

It is often impractical to visit the nodes physically in order to reprogram them [Lev02]. Therefore, several methods have been proposed to reprogram nodes through the network, by sending code over the network. This paper gives an overview of these methods.

In section 2, we introduce a simple method to reprogram nodes. In section 3, we describe a method to securely program nodes which adds authentication to the process. Some optimizations on transferring the whole code image are described in section 4. In section 5, we explain why virtual machines are useful in this context. We conclude with ideas for future work in section 6 and the conclusion in section 7.

2 Propagating code

In this paper, it is assumed that a special node called the *base station* has the new code image. To reprogram the network, this code image has to be sent to all other nodes. Not all nodes may be within radio range of the base station, so a method is needed to let other nodes pass on data.

The whole code image may not fit in one network packet. If this is the case, the code image has to be split up at the base station and reassembled at each node. Each node needs all parts of the code for the reprogramming to work. These parts of code are called *segments* in this paper.

In typical nodes, communication is expensive in terms of energy. Sending a single bit can consume the same energy as executing 1000 instruc-

tions. [Reij03, Sta03, Lev02]

In [Sta03], a protocol is proposed which uses a ripple method to disseminate code. All source nodes broadcasts code segments. Surrounding nodes store these. Any node which has all segments, and thus a complete code image, becomes a source node. According to [Sta03], this ripple protocol reduces traffic 60–90%, compared to flooding.

This has as advantage that when a segment is lost, the source which has this segment is only one hop away. Each node keeps track of which segments it has received in a sliding window. Like in TCP, the sliding window keeps track of the segments which are currently being received, but does not store information on all segments prior to the beginning of the window, which are already received. When a node detects that it has missed a segment, it asks the sender to retransmit the segment. [Sta03] does not describe what to do when this source is not reachable, although they propose to use a source-discovery mechanism.

3 Securely propagating code

Particularly in military applications, it may not be desirable for anyone else than the owner to reprogram a WSN. Therefore, the base station should authenticate itself and sign its code. However, with the limited resources of nodes, public key schemes should be used sparingly. Using a global shared secret is not safe enough, because an attacker may compromise a node and capture the key.

[Jin06] proposes to use a *hash chain*. Only the first packet is signed with a private key. Each packet then contains a code segment and the hash of the next packet. Because the first packet is signed, that segment and hash are correct. If the second packet matches the hash, the second segment and the hash of the third packet are correct, and so on.

This method successfully secures the code and

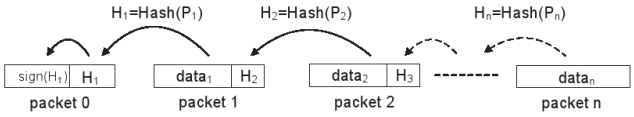

Figure 1: Hash chain: each packet contains the hash of the next packet.

verifies each packet instantly. This way, packets do not have to be stored awaiting verification, which an attacker can use to do a DoS attack. However, the packets must arrive in order, something not needed by protocols as described in section 2.

To solve this problem, [Jin06] changes the protocol to make use of a *hash tree*. With a hash tree, each packet contains w hashes. Any of these w packets can be verified, which in turn can verify other packets. This way, a tree of packets is built which verify each other. By sending packets by traversing the tree breadth-first, the node has a relatively long period to recover the lost packet. This method does not allow the packets to arrive in *any* order, but would work well when the packets are slightly out of order.

4 Only propagating updates

Since nodes are already programmed before deployment, programming after deployment is likely to consist of small changes. Fixing a bug or adding a feature often leaves most of the code intact. By only sending the changed code, communication can be greatly reduced.

4.1 Diff-like approach

In [Reij03], a method is proposed to send only the differences between two versions of the code. Instead of transmitting code, an *edit script* is sent. This edit script describes how the new version can be made from the old version and the information in the edit script. Such a method is already widely

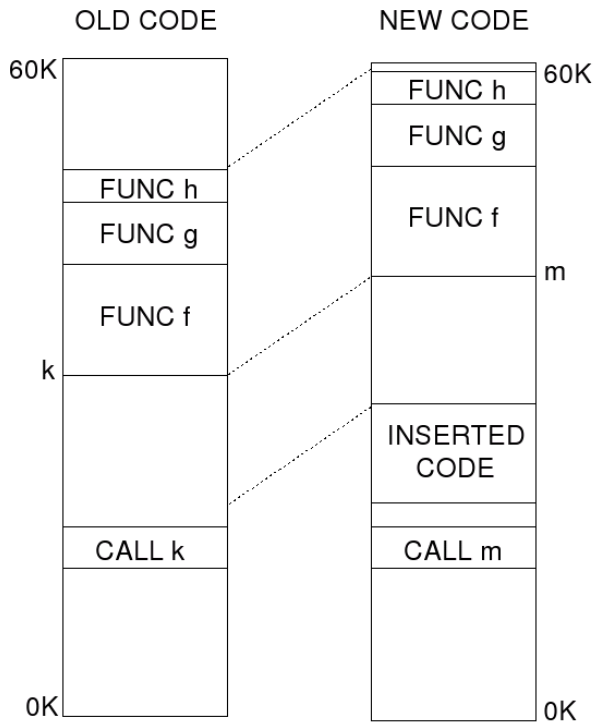

Figure 2: Address shift: when a small change is made, addresses throughout the whole program change.

used in distributing source code on the Internet, using the Unix `diff` and `patch` tools.

The authors observe that a small change in the source code often means many changes in the binary code image, because addresses shift. When the length of a function early on in the code is changed, all addresses of succeeding functions and data change. Any reference to these functions and data must also change.

To optimize this, a special instruction is added to the edit script which shifts all references in a specific code block with a certain amount. This way, address shifts can be compensated.

4.2 Loading modules

Some operating systems designed for nodes are able to load and unload programs. These operating systems often have limited functionality in their kernel and can load programs dynamically.

In the SOS operating system ([Han05]), it is possible to load modules, which are very much like programs. The modules can interact with each other with direct calls, kernel calls or message passing. When a module initializes, it has to register its public functions and their addresses with the kernel. This way, when another module calls one of these functions, the kernel knows where it is.

Whereas the function calls in SOS are resolved when the call is made, the Contiki operating system ([Dun04]) uses a relocation function which updates programs to refer to the correct addresses.

In [Dun06], the authors observe that when distributing pre-linked[1] modules, installation fails when not all nodes have the same code image. The new module references functions and data outside the module. Normally, these references are resolved by the relocation step, which is performed at runtime. However, this does not work when the node has a different code image than which the module was linked with.

[Dun06] proposes to transmit dynamic modules and link these on the node. The authors implemented a dynamic linker on top of the Contiki operating system. They successfully linked programs on a node, showing that dynamic linking is possible even on resource-limited systems.

5 Virtual machines

In order to reduce the size of the code image which has to be transferred when an update occurs, virtual machines can be used. Because a virtual machine can be relatively high-level, the bytecode[2] which is needed to program a virtual machine can

[1] Pre-linked programs are linked at compile-time.

[2] Bytecode is a binary representation of an executable program designed to be executed by a virtual machine.

be orders of magnitude smaller than native code. [Dun06, Lev02]

The biggest downside of virtual machines is that they incur significant overhead. This means that running a program on a virtual machine takes more processing time and power than running native code. An instruction on the Maté virtual machine can take between 1.03 and 33.5 times as long as in native code. Although the reduced code size saves energy, the overhead of the virtual machine consumes extra energy.

To reduce the overhead of the virtual machine, but keep the flexibility and small code size, [Lev05] proposed application specific virtual machines. These virtual machines have big parts of the application as virtual instructions, making the program even smaller and more high-level, while implementing most of the program in native code. This makes the program both efficient and flexible.

In [Wir06], a more dynamic and numerical approach is taken to use both virtual machine and native code. A network is equipped with a node which can do just-in-time (JIT) compiling. Ordinary nodes do not have the resources to perform this task, so the JIT compilation service is implemented by a more capable node. The JIT compilation service will compile a specified method into native code and broadcast it. The nodes use distributed method profiling to identify any *hot methods*, methods which take up most of the resources. These methods are compiled by the JIT compilation service and reloaded as native code. This way, a good balance between bytecode and native code is reached.

6 Future work

In this overview we described several methods to decrease the size of the code image. In section 4, we discussed a diff-like approach and several module-based approached. However, these could also be combined to only send changes in a module [Han05]. Another optimization could be to compress code using lossless data compression [Dun06].

Several of the operating systems and virtual machines discussed in this paper have room for optimizations and feature enhancement. An optimization can be to reduce the power consumption. A feature enhancement would be to implement a garbage collector. [Bal06]

Many of the proposed methods are untested in large networks, as these are simply not available. Power and profiling measurements in real situations can help to improve these methods.

Using both native code with interpreted bytecode can offer both speed and flexibility. Future research can focus on finding the optimal ratio between native code and bytecode.

7 Conclusion

In wireless sensor networks, remote updating of the nodes is essential. Because communication is relatively costly in terms of power, many of the approaches discussed in this paper focus on reducing the size of the code. A typical approach is to use a virtual machine or an operating system to accomplish this.

References

[Wir06] *Balancing Computation and Code Distribution Costs: The Case for Hybrid Execution in Sensor Networks*, Ingwar Wirjawan, Joel Koshy, Raju Pandey, Yann Ramin. 2006.

[Dun04] *Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors*, Adam Dunkels, Björn Grönvall, Thiemo Voigt. 2004.

[Han05] *A Dynamic Operating System for Sensor Nodes*, Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler and Mani Srivastava. 2005.

[Lev02] *Maté: A Tiny Virtual Machine for Sensor Networks*, Philip Levis and David Culler. 2002.

[Lev05] *Active Sensor Networks*, Philip Levis, David Gay, and David Culler. 2005.

[Lev04] *Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks*, Philip Levis, Neil Patel, David Culler, and Scott Shenker. 2004.

[Reij03] *Efficient Code Distribution in Wireless Sensor Networks*, Niels Reijers, Koen Langendoen. 2003.

[Sta03] *A Remote Code Update Mechanism for Wireless Sensor Networks*, Thanos Stathopoulos, Tyler McHenry, John Heidemann, Deborah Estrin. 2003.

[Bal06] *Multi-level Software Reconfiguration for Sensor Networks*, Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, Mani Srivastava. 2006.

[Dun06] *Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks*, Adam Dunkels, Niclas Finne, Joakim Eriksson, Thiemo Voigt. 2006.

[Jin06] *Secure Code Distribution in Dynamically Programmable Wireless Sensor Networks*, Jing Deng Richard Han Shivakant Mishra. 2006.