Patching a Patch - Software Updates Using Horizontal Patching

Milosh Stolikj, *Student Member*, IEEE, Pieter J. L. Cuijpers, and Johan J. Lukkien, *Member*, IEEE

Abstract — *This paper presents a method for optimizing software updates of consumer electronic devices running multiple applications with a common software component, called horizontal patching. Instead of using separate deltas for patching different applications, the method generates one delta from the other. Due to the large similarities between the deltas, this horizontal delta is small in size. Experimental results on two test sets, consisting of software updates for sensor networks and smart phones, show that significant improvements can be achieved. Between 27% and 84% data can be saved from transmission, depending on the type of applications and shared components*[1].

Index Terms — software update, remote reprogramming, horizontal patching, heterogeneous networks.

I. Introduction

Today's consumer applications are running on multiple, networked devices. Furthermore, in order to decrease development cost and improve interoperability, there is a clear tendency to build CE devices on top of a common platform, including an operating system [1], a middleware [2] or a virtual machine [3]. The range of these devices varies, from different generations of smart phones to home entertainment systems [4]. A more pervasive type of system is the upcoming adaptive ambient lighting system [5], which employs a network of low capacity nodes with different roles. For instance, while some nodes measure luminance, others are responsible for switching the light actuators.

Updating software is an essential feature of modern CE devices, for the purpose of bringing new functionality, or correcting discovered bugs. Since the number of devices to be updated can be large, the communication medium has limitations and the update should be swift, a software update is a non-trivial task. This is especially true for networks of embedded systems that depend on relatively small batteries.

Software is most effectively updated in an incremental fashion (Fig. 1) [6]. Incremental updates use small scripts called *deltas* (Δ), which contain instructions and data to produce an updated version from a previous one. Deltas are platform and application specific, i.e. a delta generated for one application for a specific platform cannot be used to update the same application for another platform or to update a different application. As a result, in networks of heterogeneous devices running multiple applications on top of a common software component, incremental updates foresee separate deltas for each combination of an application with a platform.

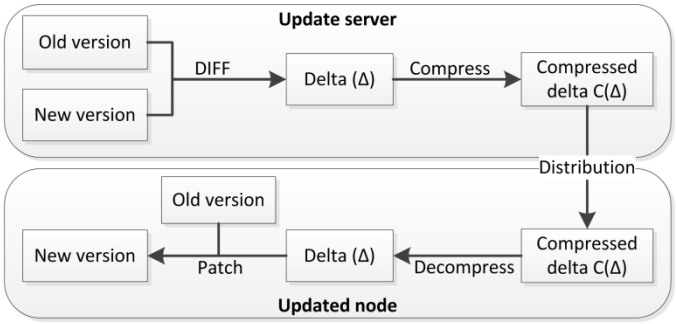

Fig. 1. Overview of an incremental update. The software update is prepared on a system which has both the old and new version of the software to be updated. Using a DIFF algorithm, the difference in the two versions is captured in the form of deltas. These deltas are usually compressed to further reduce in size, and transferred to the device which requires the update. There, after decompression, the delta is used to patch the old software version into the new version.

Given a network of heterogeneous devices running multiple applications on top of a common software component, current approaches for software update disseminate separate deltas for each device. A feasible method to minimize the communication volume is to disseminate the updates using a broadcast based scheme instead of unicasting (Fig. 2). However, the amount of data that needs to be distributed can be further reduced by exploiting the information redundancy present in the deltas.

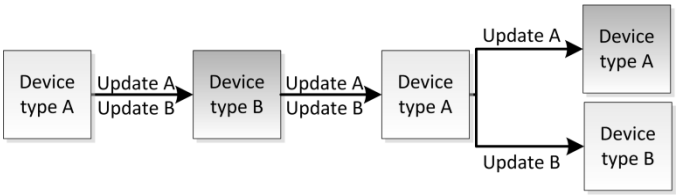

Fig. 2. Updating two different devices in a network. Even though both devices might share components, currently individual updates are prepared for each of them. Since multiple devices of the same type exist in the same network, broadcast/multicast dissemination is preferred to unicast. As a result, both updates for device types A and B reach most/all devices in the network.

This paper presents horizontal patching, a new method for handling code differences in systems running multiple applications on top of a common software component. Instead of distributing separate deltas for updating individual

[1] This work is supported in part by the Dutch P08 SenSafety Project, as part of the COMMIT program.

Milosh Stolikj, Pieter J. L. Cuijpers and Johan J. Lukkien are with the Department of Mathematics and Computer Science, Eindhoven University of Technology, den Dolech 2, 5612 AZ Eindhoven, The Netherlands (e-mail: m.stolikj@tue.nl, p.j.l.cuijpers@tue.nl, j.j.lukkien@tue.nl)

applications, horizontal patching can be used to generate one delta from the other one, so that both deltas can be distributed together. The horizontal delta is smaller in size; hence less data needs to be transmitted, saving work, bandwidth and energy.

Experimental validation on two different test sets using two different algorithms for incremental update shows significant improvements. When small devices are updated, such as software in sensor networks of low capacity devices, the resulting horizontal delta can save from 27% to 80% data from transmission. The improvements depend on the number and type of applications which need to be updated. When the operating system of smart phones is updated, between 49% and 84% data is saved from transmission. Both test cases confirm that horizontal patching can be used to great effect to improve software updates in large networks of devices sharing a common software component.

The remainder of this paper is organized as follows. Section II gives an overview on related work on software updates in multi-application networks. Section III describes the model for incremental updates and the internals of such algorithms. Section IV covers horizontal patching and its application. Section V analyses the obtained results. Finally, Section VI gives conclusions and ideas for future work.

II. Related Work

In this section, first an overview of algorithms for incremental update is given. Next, the application of incremental update in consumer electronics is covered. Subsequently, approaches for incremental updates which consider multiple deltas are analyzed and compared to the new method presented in this work.

A. Algorithms for incremental update

Incremental update uses DIFF algorithms for extracting the difference between two consecutive software versions. A rough classification of these algorithms can be made based on the type of matching done between the software versions. One group of algorithms, including VCDIFF [7] and RSYNC [8], find completely identical blocks between the two consecutive software versions. This makes them relatively fast during both delta generation and patching. The second group of algorithms, such as BSDIFF [6], find similar but not completely identical data blocks between the two software versions. Deltas generated using this approach are generally smaller, but take significantly more time to be created.

B. Application of incremental update in consumer electronics

Algorithms for incremental update are general enough to have been applied in many domain-specific applications. This ranges from software updates in mobile phones [9], sensor networks [10], on-vehicle information devices [11] etc. Domain-specific variations of the algorithms [12][13][14] have been built to enhance the delta generation process in order to further reduce the update size. These algorithm adaptations can be seen as best practices, which can be transferred to other domains to reap similar benefits.

Updates are generally disseminated in a point-to-point fashion, where CE devices connect to a central server and retrieve updates. However, when multiple devices are located in the same network and have to be updated at once, broadcast-based schemes significantly reduce the number of transmissions in the network [15].

C. Multi-version software update

Related work on multiple deltas mainly focuses on incremental updates of a single application. Kiyohara *et al* [11] enable merging of multiple consecutive VCDIFF deltas for one application to decrease the cumulative delta size. The result is a single delta which contains instructions and data to build the latest software version from any of the previous ones. The work presented in this paper is complementary, focusing on situations where multiple applications need to be updated.

Bissyandé *et al* [16] describe an epidemic propagation protocol to handle the distribution of multiple deltas of applications for the mobile operating systems. The protocol assumes that a single application can evolve into multiple orthogonal versions, hence multiple deltas exist for it. Their approach optimizes the gathering of deltas in an opportunistic fashion. Shamsaie *et al* [17] perform off-line planning of updates of multiple applications by examining which combination of deltas has the smallest size. The method presented in this paper broadens the scope of these two works, by allowing one delta to be the source of another delta, essentially expanding that search space.

The work presented in this paper is an extension to the horizontal patching approach presented by Stolikj *et al* [18]. The contributions are three fold. Firstly, a systematic approach to horizontal patching is given. Secondly, an analysis of its scalability is presented. Finally, the approach is validated on two sample sets consisting of updates of small resource constrained devices as well as updates of smart phones.

III. Incremental update in multi-application systems

Consider a network of multiple, heterogeneous devices sharing a common software component, such as an operating system, software middleware or virtual machine engine. The software stack on each device consists of a set of applications running on top of the shared software component. In certain environments, e.g. a sensor network, an embedded system, or a mobile platform, the applications are bundled and distributed together with the shared software component. In such systems, an update of the shared software component results in an update of the entire bundle. The bundle of an application with the shared software component defines a software entity for update, and is from here on referred to as the software bundle for a specific device type.

Let $S = S_0, S_1, \dots, S_{n-1}$ be the old version of the software bundle for device type i, $i = 0,1, \dots, n-1$. The new version of the software bundle is then $S' = S'_0, S'_1, S'_2, \dots, S'_{n-1}$; $i =$

$0,1,\ldots,n-1$. A simple method to reduce the size of data for transmission for update is to compress the new software before distribution:

$$C(S_i') = compress(S_i'), i = 0,1\ldots,n-1. \quad (1)$$

Algorithms for incremental update, such as BSDIFF and VCDIFF, extract the difference between the old and new software versions in scripts called deltas. These deltas hold instructions and data for how to build the new version from the old one, through a method called *patching*. Since these deltas are used to transform different versions of the same software set, they can be referred to as vertical deltas, formally defined as:

$$\Delta_i = diff(S_i, S_i'), i = 0,1\ldots,n-1. \quad (2)$$

A. BSDIFF Delta Encoding

BSDIFF is a well-established algorithm for delta encoding. An update with BSDIFF is created in two steps (Fig. 1). First, a delta (Δ) between the two versions is constructed. Then, the delta is compressed (*compress(Δ)*) and sent to the device for update. There, after decompression, the delta is applied to the old version to reconstruct the new version.

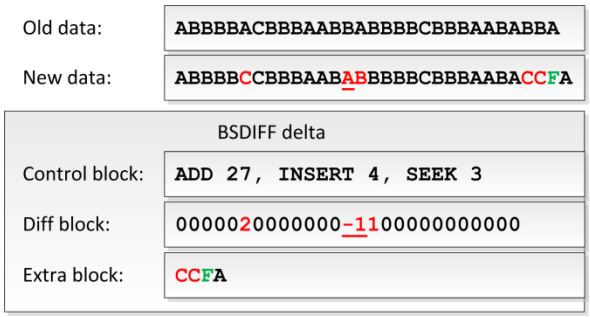

Fig. 3. Example of a BSDIFF delta. ADD specifies that the first 27 bytes from the old data and from the Diff block are summed. Zeroes in the Diff block mean that the corresponding byte from the old data is unchanged. INSERT adds four bytes from the Extra block to the output. SEEK moves the pointer in the old data three places forward, to the end of the stream.

BSDIFF has a two-pass algorithm to construct optimized deltas. In the first pass, completely identical blocks are found in the two versions. Next, these blocks are extended in both directions, such that every prefix/suffix of the extension matches in at least half of its bytes. These extended blocks correspond to the modified code.

The BSDIFF delta is built of three parts (Fig. 3): a control block of commands; a diff block of bytewise differences between approximate matches and an extra block of new data. When the old and new versions are similar, the diff block consists of large series of zeroes, which are easy to compress.

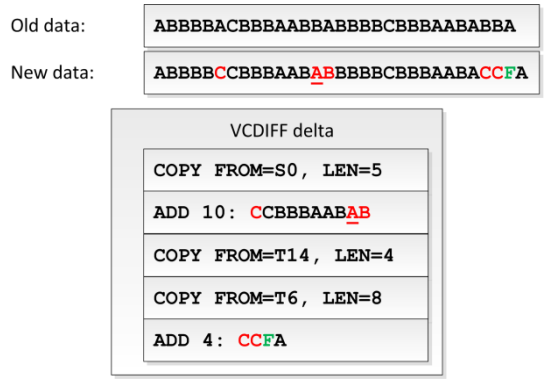

Fig. 4. Example of a VCDIFF delta. The first instruction copies the first 5 bytes from the old data (S0). Then, the next 10 bytes are added, after which two blocks from the newly written data are copied (T14 and T6). The last four bytes are again added from the delta.

B. VCDIFF Delta Encoding

VCDIFF is a format for encoding the difference between two data sets (Fig. 4). The original idea for it comes from data compression algorithms - the old and new version are concatenated; then the resulting stream is compressed using a data compression algorithm. From the output, the first part, which corresponds to the old version, is omitted, leaving only the instructions for the decoder to decompress the new version. VCDIFF features a detailed byte-code instruction set, consisting of a small number of instructions, which can be used in different addressing modes, accessing both the old and the new data. Additionally, a cache of recent addresses is held in memory.

Several tools for generating VCDIFF deltas are available. In this work, Xdelta [19] is used as an encoder for generating VCDIFF deltas. It uses additional heuristics for optimizing the generated instruction set, such as removing completely covered instructions and combining small instructions into one, essentially reducing the delta size.

IV. Horizontal Patching

Vertical deltas are not universal: a delta created for one application on a certain platform cannot be applied on a different application or a different platform Therefore, updating multiple devices in a network would require distributing each of the individual deltas, as shown in Fig. 2.

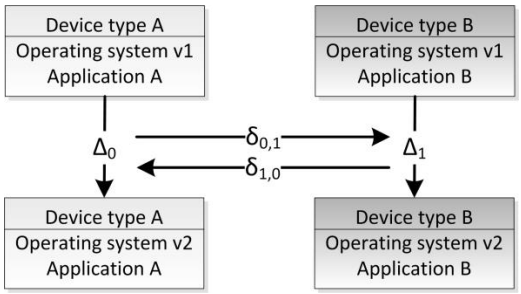

Fig. 5. Possibilities for horizontal patching in a two-application network. The two devices are sharing the same operating system.

Horizontal patching is a way to reduce the size of data that

needs to be distributed in the network. When a shared component is updated, all vertical deltas essentially hold the same information. Therefore, it is possible to use one vertical delta as a basis, and generate another delta from it (Fig. 5):

$$\delta_{i,j} = diff(\Delta_i, \Delta_j); i, j = 0,1, \dots, n-1. \quad (3)$$

Since both deltas hold the same modifications, the horizontal delta between them is smaller in size than the vertical one. The combined delta then consists of the basis and the horizontal delta ($\Delta_0 + \delta_{0,1}$ or $\Delta_1 + \delta_{1,0}$)). E.g., when Δ_0 and $\delta_{0,1}$ are used, only Δ_0 needs to be executed for updating device type A. On device type B, first $\delta_{0,1}$ is executed on Δ_0, producing Δ_1; finally, Δ_1 is executed (Fig. 6). The savings in space by using the combined delta in the multi-hop part of the network outweighs the loss in using it in the last-hop part.

All algorithms for incremental update use some form of compression to reduce the size of the vertical deltas. Unfortunately, due to the relocation and in some cases, obfuscation, introduced by this compression, it is very difficult to compute efficient horizontal delta directly on compressed vertical deltas. Therefore, we compute the horizontal deltas on uncompressed vertical deltas, and afterwards compress them for distribution.

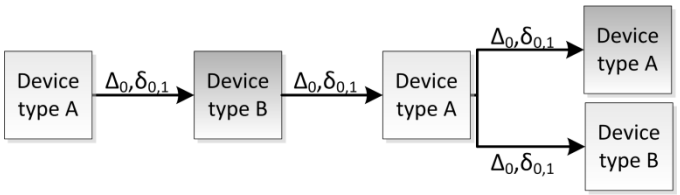

Fig. 6. Horizontal patching in practice. The basis vertical delta (Δ_0) is distributed along with the horizontal delta ($\delta_{0,1}$). On devices of type A, only Δ_0 is used for patching. On devices of type B, first Δ_1 is built by applying patch $\delta_{0,1}$ on Δ_0. Then, Δ_1 is used to patch the system.

A. Scalability analysis

The number of horizontal patches rapidly grows as the number of device types increases. Selecting the best option for horizontal update can be seen as finding the minimal spanning tree in a labeled di-graph (Fig. 7). Each vertex in the di-graph represents a vertical delta, whereas each edge corresponds to a horizontal delta. According to Cayley's formula [20], the number of spanning trees on *n* labeled vertices is n^{n-2}. For the number of possibilities for horizontal patching, this value needs to be multiplied by *n*, for each vertical delta as the base. The cost of each edge is equal to the size of the associated horizontal delta. Therefore, choosing an optimal horizontal delta would result in searching for the minimal cost spanning tree between n^{n-1} possible trees.

The processing time for finding the minimal spanning tree can quickly explode as the number of types of devices increases. This is a result of the large number of trees which have to be searched, as well as the processing time required to create the entire di-graph. The time required for building a delta depends on the size of the old and new versions. BSDIFF creates a delta in *O((x+y) log x)* time, where *x* is the size of the old version and *y* is the size of the new version. Processing time can be extremely long for large input, such as firmware images for smart phones, ranging from a few hours to several weeks. Building a complete di-graph would require computing *n(n-1)* horizontal deltas in addition to the *n* vertical deltas, which can become excessively long.

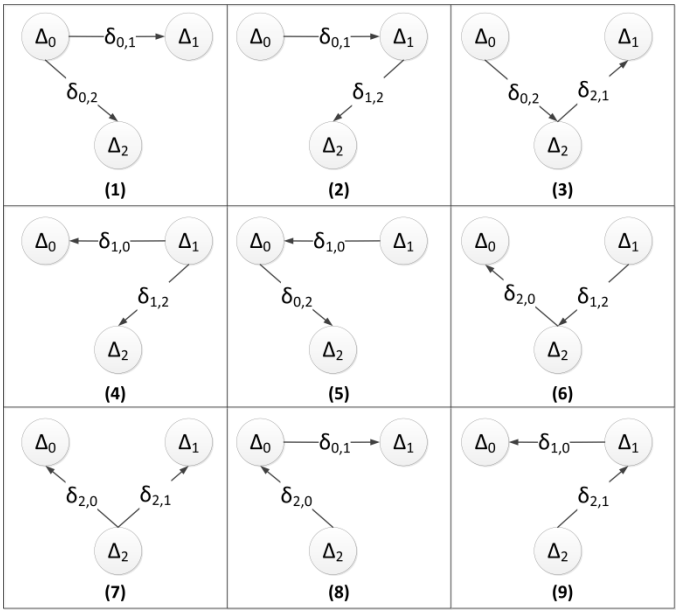

Fig. 7. Nine different options (minimal spanning trees) for horizontal patching of three heterogeneous devices.

Require: - inputs –
- number of vertical deltas N,
- array of sizes of vertical deltas Δ,
- matrix of sizes of horizontal deltas δ,

Ensure: -output -
- returns the minimal horizontal delta combination of the given di-graph,

Requires:
- edge – object with elements *from*, *to* and *weight*,
- max – returns the index of the largest element in an array.

```
1.   reachable:={};
2.   path:={};
3.   s:=max(Δ);
4.   append(reachable, s);
5.   append(path, new edge(s, s, Δs));
6.   while len(reachable)<N do
7.     min:=MAX_INT;
8.     for each i ∈ reachable do
9.       for j:=0; j<N; j:=j+1 do
10.        if j ∉ reachable and δi,j < min then
11.          min:=δi,j;
12.          from:=i;
13.          to:=j;
14.        fi;
15.      end for;
16.    end for;
17.    append(reachable, to);
18.    append(path, new edge(from, to, min));
19.  od;
20.  return path;
```

Fig. 8. Algorithm for greedy search of a horizontal delta.

In order to reduce processing time, the di-graph has to be pruned. An easy way to approach this problem is to greedily

build the tree, using the minimum number of edges for comparison. The greedy algorithm, shown in Fig. 8, expands the tree from the largest vertical delta. The largest vertical delta is chosen as a root because it requires less bits to omit data in horizontal deltas than to add new data. As a result, horizontal deltas from larger to smaller vertical deltas are most likely to be smaller in size than the reversed.

In each iteration, a new edge is selected based on two criteria: a) it connects a new vertex; b) no other edge exists such that it connects a new vertex and it is smaller in size than the selected edge. The greedy approach reduces the graph to $\frac{n(n+1)}{2}$ edges, which is feasible to compute. The performance of the greedy algorithm compared to the minimal cost spanning tree is evaluated in the next section.

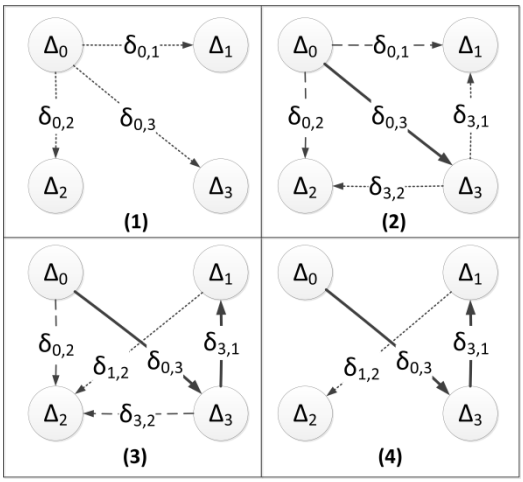

Fig. 9. Greedy search of a horizontal delta. Assuming that Δ_0 is the largest vertical delta, in the first iteration the three horizontal deltas are inspected. In (2), after $\delta_{0,3}$ is chosen, the di-graph is expanded with edges from Δ_3 which reach new vertices ($\delta_{3,1}$ and $\delta_{3,2}$). The edges towards unreachable vertices from the previous step are also taken into consideration ($\delta_{0,1}$ and $\delta_{0,2}$). After $\delta_{3,1}$ is selected (3), one more edge is computed ($\delta_{1,2}$). By adding $\delta_{1,2}$, all vertices are reachable. The horizontal delta then consists of $\delta_{0,0}$, $\delta_{0,3}$, $\delta_{3,1}$, $\delta_{1,2}$.

V. PERFORMANCE EVALUATION

In this section the performance of horizontal patching is evaluated. The first part describes the experimental setup and the two test sets used in the experiments. The second part presents the comparison between horizontal and vertical patching, as well as the difference between the greedy approach to horizontal patching compared to the optimal horizontal patch.

A. Experimental setup

The performance of horizontal patching is evaluated in a scenario for updating all devices in a network using a broadcast scheme for distributing updates. All devices are of different type, i.e. for each device a separate vertical update is used. The number of devices in the network depends on the test set being used. The first test set is for updating the firmware of sensor nodes and the second test set is for updating the operating system of smart phones.

Compression ratio, i.e. space savings, is used as a comparison metric. It shows the percentage of data saved from transmission from the original data, and is computed as:

$$compression\_ratio = (1 - \frac{new\_size}{old\_size})100. \quad (4)$$

The sum of the compressed new software bundles $\sum_{i=0}^{n-1} C(S'_i)$, is used as a reference point (*old_size*). For each combination of two or more different devices, all vertical deltas are computed with both BSDIFF and VCDIFF. Then between each pair of vertical deltas, the horizontal deltas are computed. In the end, two pairs of compression ratios are compared: vertical deltas and horizontal deltas for BSDIFF and VCDIFF.

The first test set consists of seven applications for the Contiki operating system [1]. They are built together with the operating system into one firmware image for commercially available sensor nodes. The test is repeated for three consecutive operating system updates, as shown in TABLE I. In all test cases, the applications are ordered by size, from largest to smallest.

The second test set consists of updates of an open source operating system for different commercially available smart phones. The devices have different hardware components, such as radio chipsets, sensors etc. Since vendors rarely maintain the software in such devices for a long time, not all versions of the operating system are available for all devices. Therefore, the sample set is broken into three subsets, in which an update from the old and new version exists for each model (TABLE II).

TABLE I
SIZE OF TEST DATA (COMPRESSED FIRMWARE IMAGE CONSISTING OF AN APPLICATION AND AN OPERATING SYSTEM) FOR SENSOR NODES, IN BYTES.

Application	Contiki 2.3	Contiki 2.4	Contiki 2.5
1	25,403	25,563	25,062
2	22,544	21,594	22,579
3	18,324	17,235	18,282
4	17,739	16,696	17,752
5	14,379	13,305	14,490
6	14,027	12,954	14,112
7	14,026	12,941	14,066

TABLE II
SIZE OF COMPRESSED FIRMWARE IMAGES FOR SMART PHONES, IN MEGABYTES. DEVICES 1, 4 AND 7 FORM SUBSET 1, FOR UPDATES FROM V1 TO V2; DEVICES 2, 3, 5 AND 6 FORM SUBSET 2, FOR UPDATES FROM V2 TO V3 AND DEVICES 2, 3 AND 8 FORM SUBSET 3 FOR UPDATES FROM V3 TO V4.

Device	OS v1	OS v2	OS v3	OS v4
1	193	239	-	-
2	191	-	240	256
3	187	-	234	248
4	169	200	-	-
5	163	-	195	-
6	163	-	195	-
7	148	175	-	-
8	-	-	256	272

B. Results

Fig. 10 shows the performance of horizontal patching using the sensor node test set. It is clear that horizontal patching provides higher compression compared to only vertical patching, both for BSDIFF and VCDIFF. The improvement is drastic with BSDIFF. Furthermore, as the number of different devices (applications) rises, the performance of horizontal patching improves, while vertical patching remains stable. For instance, whereas the average compression ratio of horizontal patching with BSDIFF grows from 56% with two different devices to 71% with seven different devices, the compression ratio of BSDIFF with only vertical deltas is approximately 42% in all cases.

Horizontal patching has similar performance in the smart phone test case (Fig. 11). Compression ratio is higher with BSDIFF, although both algorithms benefit from horizontal patching compared to only vertical patching. It is important to note that due to the large size of the uncompressed deltas, BSDIFF was unable to produce five horizontal deltas in reasonable time. Therefore, the number of available samples for BSDIFF for this test set is much lower.

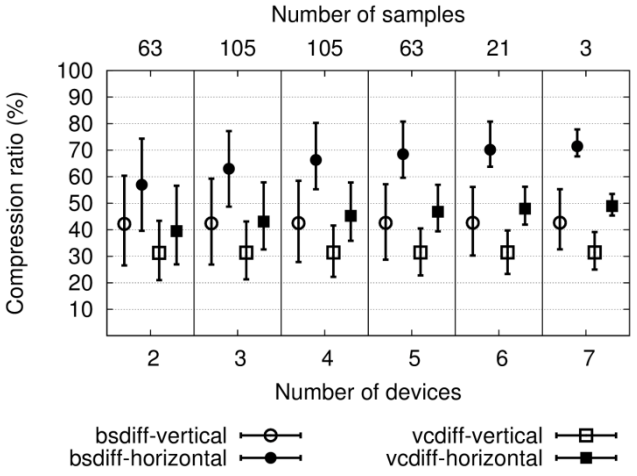

Fig. 10. Compression ratio of horizontal patching for sensor nodes using BSDIFF and VCDIFF, in comparison to compressed firmware images. Since seven applications in total are available, for three operating system updates, the number of samples available is $\frac{3 \cdot 7!}{k!(7-k)!}, k = 2,3..7$. BSDIFF has better compression ratio in all cases, although it is considerably slower compared to VCDIFF. In both cases, the compression ratio gained using horizontal patching increases with the number of different devices.

Fig. 10 and 11 show the performance of the best (optimal) horizontal delta. As previously stated, in order to find the optimal delta, horizontal deltas between all possible combinations of pairs of verticals deltas have to be generated, which consumes a lot of time. On the other hand, the greedy algorithm for building the horizontal delta is not far off the optimal one. As shown in TABLE III, the difference between the greedy approach and the optimal one is very small. The maximum measured offset in compression ratio was 1.5%, which is negligible. Therefore, the greedy approach is a sufficient solution to solve the problem of intractability of horizontal patching for large input data and high number of different devices for update.

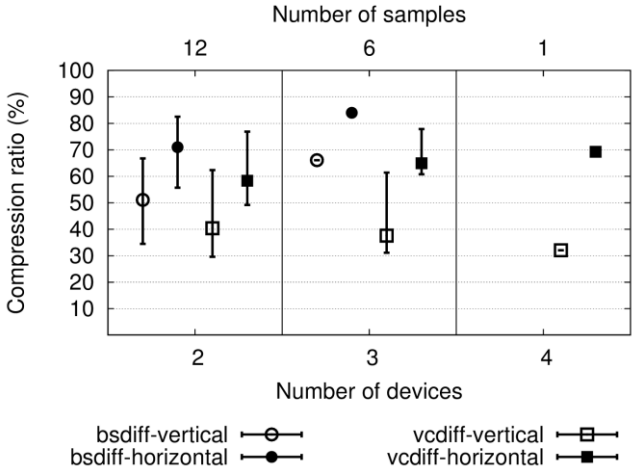

Fig. 11. Compression ratio of horizontal patching for smart phones. It is important to note that we were not able to generate five horizontal patches, therefore the number of samples for BSDIFF for two devices is six, whereas only one sample with three devices was available.

TABLE III
DIFFERENCE IN COMPRESSION RATIO BETWEEN OPTIMAL HORIZONTAL DELTAS AND GREEDY HORIZONTAL DELTAS.

Sample set	Number of devices	BSDIFF		VCDIFF	
		Average difference	Standard deviation	Average difference	Standard deviation
Sensor nodes	2	0.008	0.036	0.014	0.036
	3	0.095	0.169	0.072	0.111
	4	0.137	0.167	0.101	0.112
	5	0.162	0.148	0.120	0.107
	6	0.191	0.118	0.128	0.101
	7	0.232	0.064	0.129	0.087
Smart phones	2	0.001	0.004	0.154	0.316
	3	0.644	-	0.443	0.645
	4	-	-	0.107	

VI. CONCLUSION

This paper presents horizontal patching, a method for optimizing the size of incremental updates in a multi-application environment where heterogeneous devices share a common software component. Horizontal patching reduces the size of updates by constructing one delta from another. Therefore, when the common software component needs to be updated, horizontal patching can be used to create a smaller delta compared to the collection of deltas for each individual device.

Horizontal patching gives better results as the number of heterogeneous devices for update grows. This comes with the cost of additional processing time required for computing all horizontal deltas. The scalability analysis shows that the number of possible options for horizontal patching quickly grows and it becomes impossible to predict the final outcome. Therefore, a greedy approach is presented, which only searches through horizontal deltas which have a root in the largest vertical delta.

The improvement of horizontal patching is confirmed by

experimental validation on two test sets – one for updating software in resource constrained devices, and the second one for updating the operating system of smart phones. The impact is evident with two algorithms for delta encoding – BSDIFF and VCDIFF. For instance, with BSDIFF, the average compression ratio of vertical patching is around 42% for two to seven different devices in the first test set, while the compression ratio of horizontal patching grows from 56% with two different devices to 71% with seven different devices. Similar results are measured with VCDIFF, with 31% compression ratio of vertical patching with two to seven different devices, and 39% to 48% compression ratio with two to seven different devices.

In all test cases, the greedy approach is shown to be very close to the optimal horizontal delta. The difference between the optimal and greedy horizontal delta is at most 1.5%, which shows that horizontal patching can be used even with a large number of different devices.

The method of horizontal patching can be easily adopted for updating any CE devices which share software components. This leads to more efficient schemes for telecom operators to upgrade fleets of smart phones, tablets, television sets etc. using smaller updates.

ACKNOWLEDGMENT

The authors thank Aleksandra Kuzmanovska and Martijn van den Heuvel for many useful discussions on the scalability and usability of horizontal patching.

REFERENCES

[1] A. Dunkels, B. Grnvall and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," Work. Embedded Networked Sensors (Emnets-I), pp. 455-462, 2004.

[2] F. Battaglia, G. Iannizzotto, and F. La Rosa, "An open and portable software development kit for handheld devices with proprietary operating systems," IEEE Trans. Consumer Electronics, vol. 55, no. 4, pp. 2436-2444, November 2009.

[3] M. Vidakovic, T. Maruna, N. Teslic, and V. Mihic, "A java API interface for the integration of DTV services in embedded multimedia devices", IEEE Trans. Consumer Electronics, vol. 58, no. 3, pp. 1063-1069, August 2012.

[4] L. C. P. Costa, R. A. Herrero, M. G. De Biase, R. P. Nunes, and M. K. Zuffo, "Over the Air Download for Digital Television Receivers Upgrade," IEEE Trans. Consumer Electronics, vol. 56, No. 1, pp. 261-268, February 2010.

[5] S. Bhardwaj, A.A. Syed, T. Ozcelebi and J. J. Lukkien, "Power-managed smart lighting using a semantic interoperability architecture," IEEE Trans. Consumer Electronics, vol. 57, no. 2, pp. 420-427, May 2011.

[6] C. Percival, "Matching with mismatches and assorted applications" Ph. D. Thesis, University of Oxford, 2006.

[7] D. Korn, J. MacDonald, J. Mogul, and K. Vo, "The VCDIFF Generic Differencing and Compression Data Format," RFC 3284 (Proposed Standard), June 2002.

[8] A. Tridgell, and P. MacKerras, "The rsync algorithm," Technical Report TR-CS-96-05, Australian National University, 1996.

[9] R. Kiyohara, and S. Mii, "BPE Acceleration Technique for S/W Update for Mobile Phones," 24th IEEE Int. Conf. on Advanced Information Networking and Applications (AINA), pp. 592-599, April 2010.

[10] C. Miller and C. Poellabauer, "Reliable and efficient reprogramming in sensor networks," ACM Trans. Sensor Networks, vol. 7, 2010.

[11] R. Kiyohara, K. Tanaka, Y. Terashima, "S/W upgrade for on-vehicle information devices," IEEE Conf. Consumer Electronics (ICCE), pp. 19-20, 2012.

[12] R. K. Panta, S. Bagchi, and S. P. Midkiff, "Efficient incremental code update for sensor networks," ACM Trans. on Sensor Networks, vol. 7, pp. 30:1-30:32, February 2011.

[13] N. Samteladze, and K. Christensen, "DELTA: Delta Encoding for Less Traffic for Apps", 37th IEEE Conf. on Local Computer Networks (LCN), pp. 212-215, October 2012.

[14] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," IEEE Conf. on Sensor and Ad Hoc Communications and Networks (SECON), pp. 25-33, October 2004.

[15] M. Gumbold, "Software distribution by reliable multicast," 21st IEEE Conf. on Local Computer Networks (LCN), pp. 222-231, October 1996.

[16] T. F. Bissyandé, L. Réveillére, J.-R. Falleri, and Y. Bromberg, "Typhoon: a middleware for epidemic propagation of software updates," Middleware for Pervasive Mobile and Embedded Computing (M-MPAC), 2011.

[17] A. Shamsaie, and J. Habibi, "Planning updates in multi-application wireless sensor networks," Symp. Computers and Communications (ISCC), pp. 802-808, 2011.

[18] M. Stolikj, P. J. L. Cuijpers, and J. J. Lukkien, "Patching a patch: Software updates using horizontal patching," IEEE Conf. Consumer Electronics (ICCE), pp. 647-648, 2013.

[19] J. Macdonald, "Xdelta - open-source binary diff," 2011.

[20] A. Cayley, "A theorem on trees," Quart. J. Math. 23, pp. 376–378, 1889.

BIOGRAPHIES

Milosh Stolikj received a B.Sc. in computer science and a M.Sc. in software engineering from the Sts Cyril and Methodius University, Republic of Macedonia. In June 2011, he started research in the System Architecture and Networking (SAN) group of the Mathematics and Computer Science department of the Eindhoven University of Technology. His main research interests are in the area of wireless sensor networks. He is a member of the IEEE Consumer Electronics Society.

Pieter J. L. Cuijpers is an assistant professor in the System Architecture and Networking Research group at the Eindhoven University of Technology in the Netherlands. He received his MSc in Electrical Engineering and his PhD in Computer Science at that same institute in 2000 and 2004 respectively. After developing a process algebra for hybrid systems (HyPA), his current research interest include the application of quantitative formal methods to cyber physical systems, with a focus on performance analysis and scheduling of distributed embedded systems.

Johan J. Lukkien is head of the System Architecture and Networking Research group at Eindhoven University of Technology since 2002. He received M.Sc. and Ph.D. from Groningen University in the Netherlands. In 1991 he joined Eindhoven University after a two years leave at the California Institute of Technology. His research interests include the design and performance analysis of parallel and distributed systems. Until 2000 he was involved in large-scale simulations in physics and chemistry. Since 2000, his research focus has shifted to the application domain of networked resource-constrained embedded systems. Contributions of the SAN group are in the area of component-based middleware for resource-constrained devices, distributed coordination, Quality of Service in networked systems and schedulability analysis in real-time systems.