Typhoon : A Middleware for Epidemic Propagation of Software Updates

Tegawendé F. Bissyandé, Laurent Réveillère, Yérom-David Bromberg, Jean-Rémy Falleri

Typhoon : A Middleware for Epidemic Propagation of Software Updates

Tegawendé F. Bissyandé
CNRS/LaBRI
University of Bordeaux, France
bissyande@labri.fr

Laurent Réveillère
IPB/LaBRI
University of Bordeaux, France
reveillere@labri.fr

Yérom-David Bromberg
LaBRI
University of Bordeaux, France
bromberg@labri.fr

Jean-Rémy Falleri
IPB/LaBRI
University of Bordeaux, France
falleri@labri.fr

ABSTRACT

Applications for mobiles devices are subject to very frequent updates for fixing security vulnerabilities, ensuring compatibility with new hardware and APIs or enhancing functionalities. Getting the new version of an application involves the download of a significant amount of data, which is not practical through low-bandwidth/high-cost links. As a consequence, mobile device users often fail to update their applications.

This paper introduces a collaborative and epidemic updating scheme to improve software updates distribution. In our approach, updates are distributed by the surrounding devices, eliminating the need for costly resources. Moreover, the packaging of these updates, which consists in delivering binary patches of the difference with a previous version, dramatically reduces the amount of data to download.

Preliminary experimental results based on real contact traces show that our approach offers an efficient selection and recovery of patches, ensuring a fast update for each participating device.

Keywords

Opportunistic networking, epidemic propagation, software patch, mobile platform, Android

Categories and Subject Descriptors

C.2.4 [**Distributed systems**]: [Distributed applications]; K.8.1 [**Application packages**]: [Data communications]

*This work was partly supported by the European project ALICANTE within the framework of the EU FP7 in ICT, under grant agreement number 248652//ALICANTE/.http://www.ict-alicante.eu

M-MPAC'2011, December 12th, 2011, Lisbon, Portugal.

1. INTRODUCTION

Nowadays, personal portable devices, which include the numerous brands of smartphones on the market, increasingly run more sophisticated software that provide the same services as desktop applications, thus making their update as much critical. However, unlike desktop applications such as Adobe Acrobat Reader and Mozilla Firefox, the majority of applications for mobile devices do not patch themselves automatically. Instead, users must regularly download the entire binary of a new release to replace the outdated one. This procedure is often bandwidth and time consuming, two resources that are highly critical in mobile devices. To address this issue, a traditional approach consists in delivering smaller pieces of data, called patches, that consist only in the differences with the previous version.

The distribution of these patches often rely on a centralized model, involving a fixed number of download servers. However, this approach, even when improved by setting up a hierarchy of mirror servers, is expensive and cannot measure up in terms of scale and performance with users' needs [1]. Moreover, in these cases, bandwidth shortage concerns as well as cellular network ISP fees can make users less proactive in keeping their software updated. Finally, a centralized distribution model is actually inefficient from a networking perspective as it leaves unused the potentially very large quantity of bandwidth available for data exchange between mobile devices [13].

An alternate distribution strategy would then be to harness the potential of the many mobile devices that can be interconnected in a peer-to-peer (P2P) system for spreading patches. Indeed, since any device that joins the network can deliver a patch as well as receive some, the system is naturally balanced and self-scalable, in addition of being more effective than a fixed number of patch servers [15].

This paper. In this paper we propose to exploit the significant number of opportunities for inter-personal networking among mobile device users to distribute software updates. We leverage on the fact that while users spend a considerable amount of time together in public places, their mobile devices are increasingly left switched on [8]. In our collaborative scheme, each device, based on the history of its encounters, is able to predict whether it stands a chance to retrieve a patch required to update one of its applications.

The main contributions of this paper are :

- We first validate our assumption that software patches can be easily constructed on portable devices and that their final size is relatively small to be transferred in a reasonable time window.
- We propose an approach to patch distribution that exploits the inter-personal networking of mobile device users. This approach is supported by an algorithm that has been devised with efficiency in mind so as to deal with various challenges in mobile networks.
- To validate our approach, we perform simulations using real traces of people moving in a conference site. We then show that our prediction scheme enables the system to be effective in disseminating software updates.

The rest of this paper is organized as follows: Section 2 describes our approach and the different functionalities of Typhoon. Section 3 details the patch selection algorithm. Section 4 evaluates our approach through a simulation. Section 5 discusses related work and Section 6 concludes.

2. OUR APPROACH

Traditional update systems such as the update systems for Microsoft Windows and Apple Mac OS Software produce signed patches and distribute them using known distribution servers. OS-related software are then automatically upgraded when the patches are installed on the system. In the absence of update centers for third party applications, new software releases are entirely downloaded and installed in loco outdated software.

To leverage on the opportunistic contacts that occur while mobile device users meet, we have designed a middleware named Typhoon that enables an epidemic-style dissemination of software patches. Each participating mobile device periodically broadcasts in its environment information about the patches it offers and parses broadcast messages from its peers to identify patches that might be useful for its updating process. In the rest of this section, we present the main characteristics of the Typhoon middleware.

2.1 Typhoon services

Software patch distribution includes the construction of patches, their advertisement, their selection and the installation process. In our approach, we assume that, from time to time, at least one user has access to the Internet via his mobile device, and initiates the update process for one of the applications running on this device.

Application installer. The Typhoon middleware intercepts the download of new releases and becomes responsible for their installation, using an *installer* that mimics traditional installers, while being able to perform other tasks such as creating patches and re-constructing application releases. Thus, before installing a new version of an application, Typhoon produces a patch that corresponds to the gap between the running version and the new one, and advertises the existence of such patch to all devices in its vicinity.

Similarly, when a device recovers a patch, the application installer sets off the reconstruction process. Using the installed version of the application and the stored patches, Typhoon manages to rebuild an equivalent of the software release that were initially downloaded from the Internet by another device.

Patch selector. Considering the rapid evolution in the design of mobile device hardware and the constant enrichment of APIs, applications software are bound to be regularly renewed so as to fully exploit the capabilities of their supporting device. This in turn leads to a proliferation of different application versions being run on users' devices. Consequently, a given device in the collaborative scheme can come across a variety of software patches for the same application. While some of these patches may be immediately dismissed -when they lead to versions that are anterior to the running version-, the others can either (i) be directly used by Typhoon to update an existing application, or (ii) be downloaded in prevision of a future update. For example, if the broadcasted patch updates from version 2 to 3 and the device runs version 1, it will not be used until the bridging patch for updating from 1 to 2 is recovered. The patch can also (iii) lead to a dead-end, when there is less opportunity of moving on once it is used. This happens when two versions of the same software have been released at close dates. Only a few users had already downloaded the first version. Their patches are therefore to be avoided.

Typhoon's *Patch selector* service mainly takes into account two challenges :

1. The selection decision must be made quickly to give a chance of recovering the desired patch before users go on different ways
2. The selection must be efficient to avoid misuses of bandwidth and energy in downloading patches that are less appropriate than other patches that can potentially be recovered in the near future.

Patch propagator. A recovered patch is stored even after it is no longer needed by the device. Thus, when peers exchange patches, they increase the number of devices that make these patches available to others, hence the *epidemic* qualifier. However, space limitations can require a purge of the Typhoon patches' store to welcome other patches that are more needed by the device. Nevertheless, in such situations, Typhoon manages to leave the removal of scarce patches as an action of last resort. This way, the *altruistic behavior* of each peer contributes in improving the efficiency of the collaboration.

2.2 Construction of software patches

Patches are built on a mobile device by generating binary diff files between the original software installed on the device and the newly downloaded release. Typhoon relies on the algorithm of BSDiff [12] which provides outputs that are relatively smaller than what competitor algorithms can offer. However, after analyzing about 50 different releases of a dozen applications for mobile platforms, we have noticed that, in most cases, changes occur on the executable files but also as additions/removals of resource files such as images. Performing a binary diff on the whole software release can therefore produce output files that are unnecessarily bigger than what can be gained with a detailed diff on each file included in the application release. Such a detailed diff is called hereafter a *structural diff* in which we compute the binary diff of the only existing files that have been modified

The experiments have been conducted with Android

in the new release. Resource files that have been added in the new version are simply copied in the update package. A manifest file then summarizes the differences between the two software versions.

Nevertheless, though the update package that is obtained using the structural diff allows to reconstitute a software version that has the same size as the target release, it is not equivalent to the new release due to possible realignment of bytes. One can check this fact by simply comparing MD5 hash values of both files. To address this issue, Typhoon creates a *correction patch* by computing the global diff between the previously reconstructed release and the true new release. In the rest of this paper we refer to patches constructed by Typhoon as *structural patches* to differentiate from the usual global patch produced by BSDiff. The comparison of patches for 11 versions of the Android *FamilySafe* application showed that with our technique, Typhoon can yield much smaller patches, between 1/2 and 1/5, than a straightforward BSDiff.

Authenticity. Security in patch distribution is a critical issue that is discussed in a large body of the literature [2]. In our approach, we guarantee the preservation of the signature of the original application in reconstructed software releases. Thus, both intentionally and unintentionally malformed patches will be rejected when the application installer fails in the verification of the signature.

3. PATCH DISSEMINATION ALGORITHM

In this section, we give more details on the dissemination algorithm used by Typhoon to select, transfer, install, store and remove patches. We assume that all participating mobile devices are equipped with short-range wireless network interfaces that allow them to regularly broadcast in their environment information related to all the patches that they have in storage. From here on, all mentioned devices are mobiles devices on which Typhoon is operational. Consequently, they all participate in the collaborative scheme for propagating software updates.

The network of peers is a set of interconnected mobile devices that run each a number of applications. The peers are referred to as nodes of a P2P network. Let $\mathcal{D}^{\mathsf{A}}_{n\to p}$ be the software patch that can be used to update a given application A from the version number n to the version number $p > n$. Each node maintains a prediction table $\mathcal{T}$ where are stored the prediction values for the node to obtain different patches. Thus, whenever a node $\mathcal{N}_{\mathsf{x}}$ informs $\mathcal{N}_1$ that it carries the patch $\mathcal{D}^{\mathsf{A}}_{n\to p}$, peer $\mathcal{N}_1$ updates in $\mathcal{T}$ the value $\mathcal{P}^{\mathsf{A}}_{n,p}$, the frequency of occurrences in the past of this $\mathcal{D}^{\mathsf{A}}_{n\to p}$, which equates, in our algorithm, the prediction value for the node to come across any other peer that can provide this patch.

Event. At the occurrence of event $E(\mathcal{N}_1 \leftarrow \mathcal{N}_2)$, when node $\mathcal{N}_1$ receives a broadcast message containing some information about the patches stored by node $\mathcal{N}_2$, (1) $\mathcal{N}_1$ parses the message and retrieves the list of patches that $\mathcal{N}_2$ can provide. Then, (2) $\mathcal{N}_1$ updates in $\mathcal{T}$ the new frequency values corresponding to the advertised patches. Finally, (3) using the information stored in $\mathcal{T}$, node $\mathcal{N}_1$ deduces the highest version number z that can be reached for each application. This version number does not necessarily represent the most recent version that application developers have released, but the highest version $\mathcal{N}_1$ has ever *heard* of. Similarly, the lowest version number a for which there are patches to update from can also be computed.

Algorithm 1: Select and recover a patch $\mathcal{D}^{\mathsf{X}}_{x_1\to x_2}$ among all patches broadcasted by other nodes

```
Input: {D^A_{a→b}, D^B_{a→c}, ..., D^A_{b→d}, D^B_{b→c}, ..., D^Z_{y→z}}
listOfPatches ← {D^A_{a→b}, D^B_{a→c}, ..., D^A_{b→d}, D^B_{b→c}, ..., D^Z_{y→z}}
updateTable(T, listOfPatches)
selectedPatch ← ⊥
A ← findApplicationToUpdate(listOfPatches)
selectedPatch ← selectBestPatch(listOfPatches, T, A)
if selectedPatch ≠ ⊥ then
    ∃ (m, n) · D^A_{m→n} ∈ listOfPatches | D^A_{m→n} = selectedPatch
    if size_of(D^A_{m→n}) ≥ storeRemainingSpace then
        purgeCacheFor( D^A_{m→n} )
    download & store D^A_{m→n}
if current_version = m ∧ D^A_{m→n} ∈ Store then
    do patching
    newVersion ← n
    while ∃ D^A_{newVersion→x} ∈ Store | x > newVersion do
        do patching
        newVersion ← x
```

Listening window. Each peer in the network keeps alive a listening window to record the events that occur during a delimited time slot. When the listening window closes, all the patches that have been advertised by neighbouring peers are processed altogether.

The objective of the update process is to allow all nodes to reach, for each application, the highest version that is currently available, or, failing this, a higher version from which there is more opportunity to get, later on, the highest version. For the procedure to be effective in a network where peers are mobile, the decision on the patch to recover must be made as quickly as possible. Furthermore, attempting to retrieve all patches advertised by other peers during the listening window is an unworkable solution. We propose an approach that consists in selecting, among advertised patches, the most appropriate for one of the peer's applications. The selection algorithm described in Algorithm 1 gives an overview of the decisions that are taken by a node in the collaborative network. In the following subsections, we review the main steps of this algorithm.

3.1 Prediction table update

The selection of a patch strictly depends on the availability of all other related patches that are required to complete the update process. Indeed, when running a given application A with version number i, it is unwise to recover a patch for updating from version number $i+1$ to $i+2$ if there is no opportunity for retrieving a patch for updating from i to $i+1$. Keeping each peer's prediction table $\mathcal{T}$ up to date is therefore an essential step of the algorithm. For a given node $\mathcal{N}_1$, it consists in re-computing, for each patch referenced in $\mathcal{T}$, the prediction value for $\mathcal{N}_1$ to be interconnected with a node that can provide this patch, by updating the frequency value of its appearance in broadcast messages processed by the node. In particular, when the size of a given patch is known to be larger than the memory space that has been set aside for storage, $\mathcal{N}_1$ considers that there is no opportunity for recovering it by setting the prediction value to 0. This way, at the occurrence of a subsequent event, $\mathcal{N}_1$ dismisses any advertised patch that depends on the previously rejected patch.

3.2 Application selection

Each node, at the end of a listening period, selects an application to update, depending on the list of available patches. This selection is based on a scheme with a selection queue containing all applications that are run by the peer. When an application is chosen, its identifier is moved to the end of the queue. All applications are therefore given a chance to be updated at some point. The simplicity of this procedure also gives a better chance to the peer to recover a patch. Indeed, using a more sophisticated, hence complex, algorithm would have introduced more latency between the reception of a broadcast message and the beginning of the download, leaving enough time for the two peers to be disconnected.

3.3 Patch selection

After selecting an application, $\mathcal{N}_1$ must determine which of its patches is the most appropriate for enabling the update process to quickly lead to the highest version number. Indeed, in order to update a given application A from version number i to the highest version number z, $\mathcal{N}_1$ must progressively acquire a sequence of patches, the best sequence being the one for which the probability to collect all the necessary patches has the highest value. Considering two version numbers x and y, the probability of being able to update from one to another is obtained by computing the shortest path through the known intermediate version numbers between x and y.

The directed graph in Fig. 1 highlights the key possibilities of sequences of patches that can be used by a node $\mathcal{N}_1$ running the version i of an application A and that needs to update to z, the highest version number in $\mathcal{T}$. $\mathcal{N}_1$ must then determine among the advertised patches, those that are of useful in its current configuration.

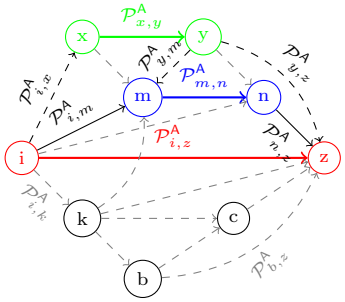

Figure 1: Subset of possible *paths* in the case where a node $\mathcal{N}_1$ which runs application A with version number i while storing a patch $\mathcal{D}^{\mathsf{A}}_{x\to y}$ comes accross a node that can offer the patch $\mathcal{D}^{\mathsf{A}}_{m\to n}$

We introduce in the following different notations and equations that formalize the computations that implement the best patch selection algorithm:

- $\underset{i\to z}{Seq}(\mathcal{D}^{A}_{k\to l})$: a sequence of known patches that constitute a possible combination to update A from i to z. We call it a possible *path*.

$$\underset{i\to z}{Seq}(\mathcal{D}^{\mathsf{A}}_{k\to l}) =< \mathcal{D}^{\mathsf{A}}_{k_0\to l_0}, \mathcal{D}^{\mathsf{A}}_{k_1\to l_1}, ..., \mathcal{D}^{\mathsf{A}}_{k_n\to l_n} > \\ | \; k_0 = i \; \wedge \; l_n = z \; \wedge \; \forall x \in]0:n], k_x = l_{x-1} \tag{1}$$

- $\cup \underset{i\to z}{Seq}(\mathcal{D}^{A}_{k\to l})$: the set of all possible paths.

- $Prob(\underset{i\to z}{Seq}(\mathcal{D}^{A}_{k\to l}))$: the overall prediction value for $\mathcal{N}_1$ to get all the patches in this path. This value is inversely proportional to the cost value that is attributed to the path in Dijkstra's algorithm.

$$Prob(\underset{i\to z}{Seq}(\mathcal{D}^{A}_{k\to l})) = \prod_{\forall (k,l) | \mathcal{D}^{A}_{k\to l} \in \underset{i\to z}{Seq}(\mathcal{D}^{A}_{k\to l})} (\mathcal{P}^{A}_{k,l}) \tag{2}$$

- $\mathsf{sPath}(\mathcal{T}, A, i, z)$: the function that computes for a given node, using the prediction table $\mathcal{T}$, the shortest path to go from version number i of application A to version number z. To implement this function we rely on Dijkstra's algorithm, using version numbers as vertices' identifiers and the inverse of prediction values as edges' costs (cf. Fig. 1).

$$\mathsf{sPath} \; (\mathcal{T}, A, i, z) = \underset{i\to z}{Seq}(\mathcal{D}^{\mathsf{A}}_{k\to l}) \in \cup \underset{i\to z}{Seq}(\mathcal{D}^{\mathsf{A}}_{k\to l}) \\ | \; \forall \; \underset{i\to z}{Seq'}(\mathcal{D}^{\mathsf{A}}_{k\to l}) \in \cup \underset{i\to z}{Seq}(\mathcal{D}^{A}_{k\to l}), \\ Prob(\underset{i\to z}{Seq'}(\mathcal{D}^{\mathsf{A}}_{k\to l})) \leq Prob(\underset{i\to z}{Seq}(\mathcal{D}^{\mathsf{A}}_{k\to l})) \tag{3}$$

- $\mathsf{aPath}_{\mathsf{x,y}}(\mathcal{T}, A, i, z)$: the shortest of all paths that include a given patch $\mathcal{D}^{\mathsf{A}}_{x\to y}$.

$$\mathsf{aPath}_{\mathsf{x,y}}(\mathcal{T}, A, i, z) = \\ < \mathsf{sPath}(\mathcal{T}, A, i, x), \mathcal{D}^{\mathsf{A}}_{x\to y}, \mathsf{sPath}(\mathcal{T}, A, y, z) > \tag{4}$$

Best path computation. The computed best path may include patches that are in storage, patches that are currently advertised, and patches that the peer predicts to retrieve in the future. It is therefore used as a reference path necessary to weigh other alternate paths. Thus, while relying on the prediction table, node $\mathcal{N}_1$ takes every concrete opportunity it gets to recover less popular patches.

Once the utility of a patch has been established, the receiving node must ensure that there is still enough memory space to welcome the new patch. In case of a memory deficit, a few decisions can be taken to purge the store by deleting progressively that are no longer needed by the node. Nevertheless, since the system is collaborative, the algorithm adds some intelligence in the purge by removing in priority patches that are popular in the network. Thus, any other node needing this patch still have a chance of retrieving it from other nodes. If the node need more space, it can begin deleting patches that have been recovered but whose use is less pressing. These patches include patches that are meant to update an application from a version j that is much higher than the current version i run by the node.

Dijkstra's algorithm for finding the shortest path produces the optimal path with minimal costs

4. ASSESSMENT

We have implemented a prototype of the Typhoon middleware for the Android platform. Spontaneous interconnections are possible using the zeroconf technology on Android "development" phones for WiFi networking. However, to support all devices on the market, our prototype implementation also integrates a customized Bluetooth OBEX service which allows to sidestep the lack of broadcast feature in the Bluetooth standard specifications.

The assessment of our approach has involved tests on popular software releases to confirm the adequacy of our technique for patch construction. Due to practical constraints, we rely on simulations to assess the patch propagation algorithm. Nevertheless, we use well-known human mobility traces to emulate realistic settings.

4.1 Patch size

Opportunistic networks are by definition volatile. While users meet and go their ways, their personal devices interconnect for a limited amount of time during which patches must be downloaded completely as a whole. Indeed, instead of resorting to splitting the update package into several patches that may be recovered from different peers, thus introducing security leaks, we focus on making the update package as small as possible to transfer data between participating devices. Using our technique (cf.2.2), we have produced *structural patches* for various versions of 12 applications for the Android platform. Our experiments showed that for minor updates, i.e. for update packages that have been constructed from consecutive releases, most patches have sizes below 100 kB. When we consider all possible combinations of versions to build minor as well as major update packages, the median size value increases to about 300 kB, which is still significantly less than the usual media files transfered in opportunistic networks. For instance, McNamara *et al* have considered MP3 files that are around 5 MB in size [7].

4.2 Simulation

To validate our approach of software patch dissemination in opportunistic networks, we conduct a simulation using real-world datasets of human mobility traces that have been extensively referenced in the literature [3, 13]. The Infocom06 [14] trace contains data collected during CRAWDAD's Experiment 6 at the Infocom'06 conference in Barcelona, Spain. The experiment has recorded the Bluetooth sightings between iMotes that have been distributed to participants, students and researchers, attending the student workshop that was co-located with the conference. In our simulation, we consider that the 78 valid IDs that have been kept by the trace's authors correspond to mobile devices that participate in our collaborative scheme, and therefore rely on Typhoon. To eliminate bias, we have added to our simulation variables a parameter that discards data related to the period of time when all the participants seem to be stuck together for a significant amount time, for example while attending the same talk. To better understand the used dataset, we perform a first round of simulation in which each node, at its turn, is given a patch of size 0 to be transfered to everyone. The null patch size allows all sightings, regardless of their duration, to be counted as useful. The median propagation curve in Fig. 2 is closer to the faster than to the slower, indicating that the majority of nodes in the Infocom06 trace will contribute to the success of an epidemic distribution model.

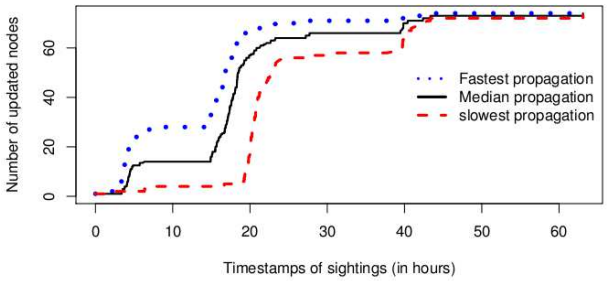

Figure 2: Users rarely get isolated in a real-world situation

Simulation settings. In all configurations of our simulation scenarios, we have fixed the value of transfer rate, and the duration of the listening window. Experiments we have conducted in our computer laboratory while many members of the staff had their Bluetooth-enabled devices switched on, show that interferences among signals can deteriorate the transfer rate. For our simulations, we consider the worst case scenario by setting the transfer rate to $25kB/s$, so as to account for all devices with the lowest connectivity.

Simulation results. We consider for the configurations of our simulation different variables that can have a significant impact on the success of the collaborative scheme. For each variable, we run a scenario that assesses its influence.

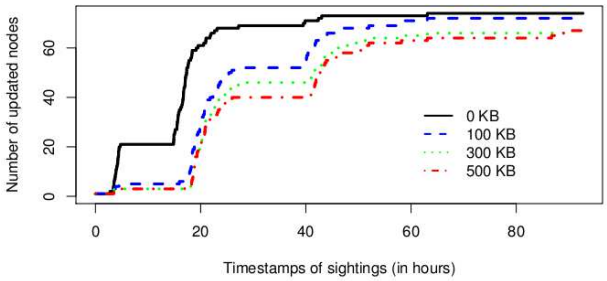

Figure 3: The bigger the patch, the slower the propagation.

❶ Impact of patch size: Given the transfer rate and the mobility of devices, the patch size is bound by the contact duration. We run a simulation where only one node provides the patch whose size is varied between 0 KB to 500 kB. Fig. 3 reveals that for the median size value of patches constructed with Typhoon, which is around 300KB, and with the transfer rate suggested, the epidemic dissemination performs well.

`http://www.zeroconf.org`

cmwrap, ConnectBot, FamilySafe, FlckrFree, FoxSaveer, GoogleMaps, ListBuddy, Ringdroid, ScrambledNet, vcardio, WebSMS, XBMCRemote

Bluetooth does not support broadcast. We emulate this capability with a dedicated Bluetooth service. Besides, since our workloads have been gathered from Bluetooth sightings, we perform the simulation with Bluetooth properties

❷ Impact of the number of providing peers: An important factor of propagation in epidemic models is the number of *infected* hosts. We run simulations where we vary the number of peers that have been able to autonomously download a new release from the Internet and produce a patch. We consider different cases where peers run a given application A, with n peers running version $v2$ while having in storage the software patch for updating from $v1$ to $v2$. The other peers run A with $v1$, and consequently must download the patch $v1 \rightarrow v2$ when they get the chance. We vary the value of n from 5% to 80% of the total number of peers. The graph of Fig. 4 shows that the propagation is the fastest when there are many nodes that propose the necessary patch.

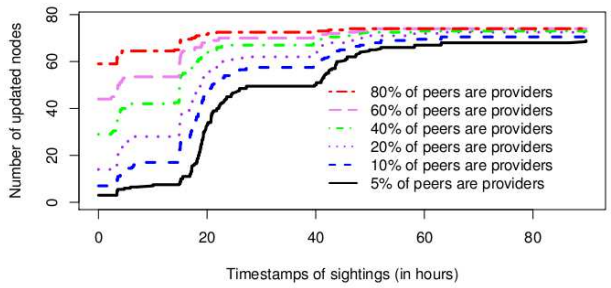

Figure 4: The more providers, the better the propagation

❸ Impact of the variety of update levels: Given that mobile devices may run different versions of the same application, different patches must be available to allow the update of all devices. In our simulations all peers run a given application A with either version $v1$, $v2$ or $v3$. Peers running $v1$ have initially no patches in storage. Peers running $v2$ have initially patches for updating from $v1$ to $v2$ ($v1 \rightarrow v2$). Peers running $v3$ have $v2 \rightarrow v3$ patches. We consider a reference configuration where there is an equilibrium in the update level, each third of all peers running A with either $v1$, $v2$ or $v3$. Then we consider other situations with variations in the proportions of the different patches. Overall, the graphs of Fig. 5 show that the algorithm we have implemented succeeds in balancing these constraints, especially through its selection algorithm which is aimed at avoiding dead ends.

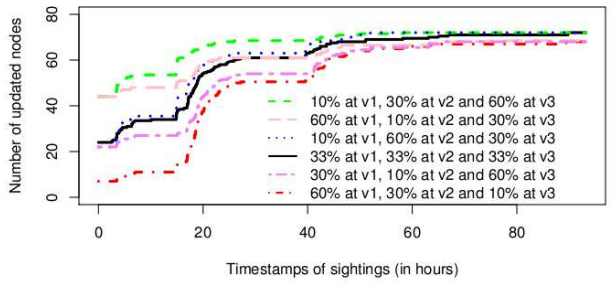

Figure 5: The more patches, the better the propagation

5. RELATED WORK

Software update distribution is of great interest for software providers who face the issue of how to keep the largest number of software updated on user's devices. IT companies mostly rely on centralized solutions that can be ineffective in terms of load balancing and bandwidth consumption.

Gkantsidis *et al.* have presented a study of various patch delivery strategies that can improve the performance of solutions used in the *Windows Update* system [4]. They show that if users stay connected to the central server a little longer after they have downloaded the patches their machines can be used as caches for other users. Our work is based on the spontaneous P2P interactions between mobile devices which can be resource-constrained. We also propose an heuristic for prioritizing update packages in the occupation of the cache on user's device. P2P networks have also been proposed by Shakottai to fight against internet worms [15]. However, they rely on a *pull* mechanism that may submerge nodes with request to process.

Epidemic models. The propagation of computer worms has lead to an upsurge of interest in the field of epidemic propagation of content in the Internet. A great deal of work has gone into analysing scenarios of dissemination of content using P2P epidemic models [16, 9]. Our work leverages on the findings that have been recorded in this field.

Opportunistic networking. Human daily activities involve strong patterns of encounters and movements. Bluespots exploit public transport infrastructures to provide a distribution system to the daily users [5]. However, the centralized hubs that are used, in addition of constituting a single point of failure, can bring out contention issues.

The *familiar stranger* [11] in big cities is a given individual that another person regularly meets and stays with in the same environment for a reasonable amount of time during which they can exchange information. With the proliferation of hand-held devices equipped with wireless technologies, researchers have shown a significant interest in using inter-personal networking for sharing information among devices [10, 6, 7]. Contrary to what is done in most projects, the data exchanged by Typhoon must be selected using different criteria. For instance, while in the approach of McNamara *et al.* a shared media file is selected according to user's preferences, an update package is selected not only if there is an application that needs it, but also if it can effectively be useful in updating this application to the highest version possible. The data itself therefore has a significant influence on the dissemination algorithm.

6. CONCLUSION

In this paper we have presented an approach to distribute software updates during opportunistic contacts. The approach is based on a prediction scheme that enables mobile devices to select and recover the best update package for speeding up the update of all running applications. We implement in the Typhoon middleware an algorithm that accounts for different factors which can influence the dissemination of patches. The most salient of these variables are (1) the size of the patch so as to readily transfer to devices that have limited capabilities, (2) the transfer rate so as to account for the volatility of opportunistic networks, and (3) the availability of participants willing to share their data. Using real-world workloads, we have established that

Typhoon's algorithms are effective in identifying the best patch, while avoiding any waste of time and resources if users are expected to recover better patches later.

Future work will involve a thorough assessment of energy consumption and the support for a *best source* selection algorithm to leverage on the opportunity of various contacts that are actually similar in that they propose the same patches.

7. REFERENCES

[1] Serge Abiteboul, Itay Dar, Radu Pop, Gabriel Vasile, Dan Vodislav, and Nicoleta Preda. Large scale p2p distribution of open-source software. In *VLDB '07*, pages 1390–1393, Vienna, Austria, 2007.

[2] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure software updates: disappointments and new challenges. In *HOTSEC '06*, pages 37–43, Vancouver, Canada, 2006.

[3] Augustin Chaintreau, Pan Hui, Jon Crowcroft, Christophe Diot, Richard Gass, and James Scott. Impact of human mobility on opportunistic forwarding algorithms. *IEEE Trans. on Mobile Computing*, 6(6):606–620, 2007.

[4] Christos Gkantsidis, Thomas Karagiannis, and Milan VojnoviC. Planet scale software updates. In *SIGCOMM '06*, pages 423–434, Pisa, Italy, 2006.

[5] Jason LeBrun and Chen-Nee Chuah. Bluetooth content distribution stations on public transit. In *MobiShare '06*, pages 63–65, Los Angeles, CA, 2006.

[6] Vincent Lenders, Martin May, Gunnar Karlsson, and Clemens Wacha. Wireless ad hoc podcasting. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12(1):65–67, 2008.

[7] Liam McNamara, Cecilia Mascolo, and Licia Capra. Media sharing based on colocation prediction in urban transport. In *MobiCom '08*, pages 58–69, San Francisco, CA, USA, 2008.

[8] Eamonn O'Neill, Vassilis Kostakos, Tim Kindberg, Ava Fatah gen. Schiek, Alan Penn, Danaë Stanton Fraser, and Tim Jones. Instrumenting the city: developing methods for observing and understanding the digital cityscape. In *UBICOMP 2008*, Seoul, South Korea, 2006.

[9] Oznur Ozkasap, Mine Caglar, and Ali Alagoz. Principles and performance analysis of second: A system for epidemic peer-to-peer content distribution. *J. Netw. Comput. Appl.*, 32(3):666–683, 2009.

[10] Maria Papadopouli and Henning Schulzrinne. Effects of power conservation, wireless coverage and cooperation on data dissemination among mobile devices. In *MobiHoc '01*, pages 117–127, Long Beach, CA, USA, 2001.

[11] Eric Paulos and Elizabeth Goodman. The familiar stranger: anxiety, comfort, and play in public places. In *CHI '04*, pages 223–230, Vienna, Austria, 2004.

[12] Colin Percival. *Matching with Mismatches and Assorted Applications*. PhD thesis, University of Oxford, 2006.

[13] Joshua Reich and Augustin Chaintreau. The age of impatience: optimal replication schemes for opportunistic networks. In *CoNEXT '09*, pages 85–96, Rome, 2009.

[14] James Scott, Richard Gass, Jon Crowcroft, Pan Hui, Christophe Diot, and Augustin Chaintreau. CRAWDAD trace cambridge/haggle/ imote/infocom (v. 2009-05-29), May 2009.

[15] Srinivas Shakkottai and R. Srikant. Peer to peer networks for defense against internet worms. In *Interperf '06*, page 5, Pisa, Italy, 2006.

[16] Wei Yang, Wei Ying, Gui-ran Chang, and Zhuo-qun Zhang. Research on the epidemic model in p2p file-sharing system. In *HIS '09*, pages 386–390, China, 2009.