FEW Phone File System

João Soares and Nuno Preguiça

CITI/Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

Abstract. Data availability is an important issue in mobile computing environments. Both distributed data management systems and mobile storage devices, such as USB-flash drives, are used to allow users ubiquitous access to their data, but both solutions have their shortcomings.
In this paper, we present the FEW Phone File System, a system designed to allow users to access their personal data independently of user's location or machine's connectivity. By combining the advantages of both mobile storage devices and distributed data management systems, the FEW Phone File System allows users to carry replicas of their data on their mobile phones, and access them as regular files in a file system, by relying on the wireless communications capabilities of these devices.
To deal with the limitations of these devices, the FEW Phone File System integrates mechanisms that allows multimedia contents to be adapted to the mobile phone's specifications, thus using much less storage. Additionally, the system includes a data source verification mechanism that keeps track of additional sources for each file (e.g. web, CVS/SVN repositories), thus allowing for these remote sources to be used as source alternatives for the user's data.

1 Introduction

Today's users are no longer confined to a single personal computer, instead they tend to have multiple personal computers. To be able to work, at any time, users can have access to their personal data by relying on distributed data management solutions (e.g. Distributed file systems [1], Internet Workspaces, etc.) or by using mobile storage devices (e.g. USB flash drives, etc.) for maintaining replicas of their personal data.

The usability of distributed data management solutions is limited by the performance of network connectivity and the difficulty to setup a personal server for the average user in an environment full of private networks, firewalls, etc. The use of mobile storage devices tends to be more popular, but it force users to keep replicas manually synchronized and handle back-up copies to deal with the possible loss, theft or failure of the device. This problem is made worst since users tend to have multiple devices (both computers and mobile storage devices), with data replicas in each one. Users are also forced to deal with concurrent data updates whenever the latest version of their data is unavailable.

The goal of this work is to develop a system that allows users to always access their work or personal data, independently of their location (computer host) or machine connectivity, by combining the strengths of distributed data management systems and mobile storage devices. Replica management issues should be automatically dealt by the system, leaving users free from these tasks. Also, the system should allow concurrent access to replicas, thus automatically dealing with conflicting updates. The system should be able to use replicas stored both in mobile storage devices and remotely to provide the best service possible.

1.1 Proposed Solution

Mobile phones, not only make part of our current day's life, but also have considerable computing power, storage space and many different wireless communication capabilities. Our solution takes advantage of the storage capacity of these devices, for maintaining replicas of the users' data, and their wireless communication capabilities, for allowing these replicas to be accessed as regular files in any computer with a wireless communication interface. This way, users will always carry their personal files with them, and will be able to access them in any computer, independently of their location and connectivity.

The FEW Phone File System (FEW Phone FS) is a distributed data management system, relying on optimistic replication, that allows users to always access their personal data. It is based on the client-server model, using the mobile phone as a portable personal file server.

To address the storage limitations imposed by these devices, the FEW Phone FS allows for multimedia contents to be adapted to the device's specifications, thus reducing the volume of data stored on the mobile phone. This also contributes to the reduction of communications with the mobile phone, while reducing the device's energy consumption.

A data source verification mechanism allows the FEW Phone FS to keep track of the additional sources of files managed by the system. This mechanism allows the system to retrieve the URLs of the resources obtained from remote sites, thus allowing direct access to these resources whenever a high-speed network connection is available. When data stored in the mobile phones is an adapted one, this mechanism also allows users to access the full fidelity versions of the data, by obtaining original contents from other sources (or nodes in the system).

1.2 Contributions

FEW Phone FS presents a solution for data availability in a mobile computing environment, while also addressing the limitations imposed by such environments. The most important aspects of its design are:

– An optimistic replication mechanism, allowing for replicas to be created and accessed when needed, including a novel scalable update tracking solution. These characteristics allows for disconnected operation.

– Reconciliation techniques that automatically detect and resolve possible conflicting updates.
– A mechanism that allows for multimedia contents to be adapted to the specifications of mobile devices, reducing storage and communication overhead.
– A mechanism for identifying the replicas of the user's data, allowing access to any of the existing replicas based on that information.

2 Design

The FEW Phone FS is a distributed data management system based on the client-server model. The mobile phone (or multiple mobile phones) acts as a portable file server for the system, maintaining the user's data, while the computers act as clients, handling file system calls made to the system. The FEW Phone FS also allows peer-to-peer interaction among clients, thus addressing power limitations imposed by the mobile phone.

The system relies on an optimistic replication design that allows for long lived replicas to be created and accessed in any client (computer), at any time. This way, client requests can be fulfilled directly from the long lived replicas stored on the local file system, thus contributing for the reduction of the mobile phone's power consumption due to communications.

Data replicas are synchronized between the client and the server by a periodic reconciliation process, whenever the mobile phone is close to a computer. As we expect the user to always carry her mobile phone, this process guarantees that the most recent version of the user's data will be kept in the mobile phone. Thus, users will always have access to the most recent version of their work, even in disconnected machines. Additionally, the system guarantees that data replicas stored in computers are synchronized as users move close to them.

The optimistic replication approach also allows disconnected operation, even when the user forgets his mobile phone. This increases the possibility of conflicting updates to occur. The reconciliation process is designed to automatically detect and resolve conflicting updates, combining the "last version wins" rule and a "no update is lost" policy, as explained in section 2.2.

The FEW Phone FS operates on sets of files known as *containers*. A container is a subset of the file system tree. It can be seen as a file system that has its root on some base directory. The reconciliation process typically operates at the container granularity, although it is possible to synchronize only a subset of the container.

The use of a mobile device as a mobile phone, forces the FEW Phone FS to deal with the limitations imposed by this kind of devices.The FEW Phone FS addresses power consumption limitations by reducing the number of tasks preformed on the mobile phone, and by reducing both network interactions with the mobile device as well as the volume of data transferred during these interactions. Storage capacity limitations are addressed by reducing the volume of data written to the mobile phone's memory. The integration of a Data Transcoding (DTC) and a Data Source Verification (DSV) module are essential for accomplishing these goals.

The FEW Phone FS is designed to take advantage of the available wireless technologies. For this purpose, it features a modular communications component that allows users to add new connectivity modules enabling the system to be further extended. The server (mobile phone) is responsible for determining which technology to use for the communications channel. The choice is based on the available technologies and the power state of the mobile phone. The decision will select the technology that presents the best performance/power consumption ratio.

2.1 Version tracking mechanism

Each replica managed by the FEW Phone FS maintains a set of attributes (metadata). These attributes allow the system to address replicas unequivocally (i.e. by their globally unique identifiers (GUID)) and also to determine the current state of each replica.

Each replica keeps, for each file, a version vector [2] that maintains a summary of the updates performed on the file. A version vector V for a file f is maintained as a set of pairs (Si, Ci), where Ci represents the number of updates performed on f, at site Si. We define $V(Si) = Ci$ when $(Si, Ci) \in V$, and $V(Si) = 0$ when $(Si, \_) \notin V$.

Unlike the common approach [3, 1, 4, 5], the FEW Phone FS uses an 'updated' flag to mark replicas as updated instead of immediately updating the version vector. This is used to prevent adding new entries to a replica's version vector, by using existing entries to record updates whenever possible. For example, consider two synchronized replicas of file 'x', one in node 'A' and the other in node 'B', with a version vector with only one entry for node 'B'. Without loss of generality, suppose the version vector entry for B is 1, i.e., the version vector can be represented by the set $(B, 1)$. With our approach, an update to the replica in node 'A' will mark that replica as updated, instead of adding a new entry to the vector (the normal approach would be to update the version vector to $(A, 1), (B, 1)$). During reconciliation, the "updated" flag signals the update, and, since the version of both replicas is the same, the vector is incremented in the entry of 'B' instead of creating a new entry for 'A'. This is possible as version vectors are used to keep track of updates, making no difference where the updates are recorded (as long as there is no concurrent updates). If concurrent update exist or no replica involved in the synchronization has an entry in the version vector, new entries are created to record the updates, as usual.

This approach allows the system to minimize the number of entries in the version vectors, thus contributing for reducing space overhead, while improving scalability. In our scenario, where the synchronization process tends to include a single or a small number of mobile phones, it is usually possible to keep just a entry for each mobile phone in the version vectors, independently of the number of replicas created in other computers.

2.2 Reconciliation

The reconciliation process is divided into two stages: directory reconciliation and data reconciliation.

Directory reconciliation allows the system to reach a coherent directory structure from two, possible conflicting, structures. For this purpose, a log is used for storing directory update information. Each node maintains a replica of this log, adding new entries every time a directory update is executed. Each log entry contains an unique entry identifier, the type of update executed, the GUID and the version of the updated object. A log entry's unique identifier allows the system to, in case of conflict, reach a consistent state during the reconciliation process. Possible directory updates are: create, delete, and rename file/directory. A version vector [2] is also associated with the log structure, and is used to store a summary of the updates produced by each node to the directory structure.

During the directory reconciliation process the log's version vector is used for determining the unknown updates between nodes. Each node's unknown updates are then exchanged and executed respecting their causal order. The convergence of both replicas is guaranteed since we have designed operations such that every pair of concurrent operations commute (see [6] for more details).

The basic idea for making concurrent operations commutative is to allow different objects to have the same symbolic name, but different GUIDs. Thus, operations over different GUIDs trivially commute. For operations on the same GUID, we implement a *write with the later timestamp* wins policy. For objects with the same symbolic name, the system deterministically adds some disambiguator to one of the objects, while internally maintaining the original name. Users are free to keep the system as it is, but they are expected to resolve the conflict by renaming one of the objects.

The data reconciliation process allows the FEW Phone FS to synchronize the contents of the stored replicas. To optimize the reconciliation process, a file update marks all directories in the file's path as updated, all the way to the file system's root. This simplifies the data reconciliation process, since it guarantees that the contents of two replicas of the same directory are synchronized when neither one is marked as updated. This way, the reconciliation process can skip those directories. The 'updated' flag is also used to avoid the propagation of update information to already marked directories, allowing this process to stop when it detects an already marked directory.

Concurrent updates are resolved by selecting the replica that reflects the update with the largest identifier using the Lamport [7] order on pair (counter, site ID). This guarantees that the same version wins, independently of the number of replicas involved. The "loser" version is moved to a special "lost and found" directory, allowing users to restore it if needed.

3 Addressing Mobile Phone's Limitations

Using a mobile phone as a portable personal file server requires using extensive storage space and communications in the mobile phone. In this section, we describe two modules of the system that help addressing these issues.

3.1 Data Transcoding Module

Currently, users tend to have large amounts of high-quality multimedia data, such as digital photographs, music and videos. In many situations, the user can satisfactorily use lower-quality version of the data. For example, an 8 mega pixels digital photograph with 32 bit color depth can only be presented with a resolution of 320x240 pixels and a color depth of 16 bit, in most high-end phones, and a 24 bit depth is often more than satisfactory in any computer.

The Data Transcoding module (DTC) is designed to convert multimedia contents to best fit the specification of the mobile phone (or other specified by the user). In the previous example, the 8 mega pixel digital photograph can be rescaled and its color depth decreased in order to best "fit" the mobile phone's screen specifications, thus reducing its data size. This leads to less storage consumption while reducing the volume of data transferred to the mobile phone.

The DTC module is used during the reconciliation process, before transferring multimedia contents to the mobile phone. Re-encoded versions are transferred instead of the original ones. This process allows users to access these contents on the mobile phone or in other computers, reducing the storage and communications required to access them. When combined with the following module, it can also allow users to access the full-fidelity version in other computers.

3.2 Data Source Verification Module

Currently, an important fraction of the files stored by users have been obtained from remote sites - e.g. the web. Additionally, an increasing number of on-line storage services allow users to rely on these services for storing or sharing personal data. These systems can be used as alternative storage sources for users to access their data.

The Data Source Verification module (DSV) is designed to check and retrieve the source(s) of a file. If a file has been transferred from a remote site (e.g. HTTP), this module is used to obtain that site's URL. This way, a well connected machine (i.e. with a high-speed network connection) may use this information for retrieving data from the remote site, rather than from the mobile phone.

The DSV module works as a network proxy, by monitoring the user's connections. Whenever a file is updated in the local disk, the system verifies if it has been obtained from a remote site, by comparing a secure hash (SHA-1) of the stored file with recently transferred contents. If the new contents have been obtained from a remote site, the URL of that site and the secure hash of the contents are added to the replica's metadata. During the reconciliation process

this information is transferred to the mobile phone together with the replica's contents (or the transcoded version for multimedia contents). When the server is synchronizing with other sites and a high speed network connection is available, the URLs can be used to transfer those contents from the remote sites, instead of using the replica stored in the mobile phone. The secure hash stored in the metadata is used to verify that the URL still references to the same contents. This approach allows to reduce communications with the server, which also reduces power consumption.

FEW Phone FS nodes can also be used as data sources for retrieving data contents. This way, each file kept in the server also maintains the URLs of the sites that store a replica of that file. Availability improves with the number of replicas in the system. The number of references per replica can be used to determine the need for maintaining those contents in the mobile phone. In the limit, the server can be used for storing only the URLs (and other metadata) of the available replicas.

This module also allows the system to access the full-fidelity versions of the stored data. Thus, transcoded versions are only transferred from the server when the original version is inaccessible. In this scenario, the transcoded contents and the URLs for the original ones are transferred to the client, and the new replica is marked as 'not original'. Whenever the replica is accessed, the system will try to retrieve the original contents from the available sources. This process is repeated until the full-fidelity version is transferred to the node, at which time the replica is marked as 'original' and the URLs are discarded.

4 Evaluation

The current prototype of the system was developed in Java. The client side was developed for Linux, relying on FUSE-J for intercepting file system calls (FUSE-J is a Java wrapper for the FUSE [8] system), and was executed on a desktop computer with the specifications presented in Table 1. The server side was executed on an Apple iPod Touch 1G.

In this section, we present some performance results obtained using our prototype. The results presented next are always the average of 8 runs, after removing the highest and lowest obtained results during those runs. All read and write tests were preformed with a clean file system cache (after the system was rebooted).

4.1 Read-Write Evaluation

The results presented in Table 2 were obtained by compressing files into an archive, thus evaluating the read performance. The archive was stored in the local file system, and was created using the 'tar -czf' command. The source files were stored on the local file system and in the FEW Phone FS.

From the obtained results it is possible to observe the reduced and constant overhead imposed by the FEW Phone FS. The overhead does not affect the usability of the system for normal operation.

Table 1. Evaluation environment specifications.

Operating System:	Linux Ubuntu 8.04.2
Kernel version:	2.6.24-23
CPU:	Intel® Core™ 2 Duo T7200 @ 2.00 GHz
RAM:	2.00 GB
File System:	ext3
FUSE version:	2.7.2
FUSE-j version:	2.4 pre-release 1
Bluetooth:	v2.0 + EDR
Wi-Fi:	802.11g

Table 2. Measured times for adding files to an archive.

Total Files Size	**Number of Files**	Adding files from		**Overhead**
		Local FS	**FEW Phone FS**	
648 $kBytes$	84 files	0.193s	0.235s	$\approx$ 22%
2.3 $MBytes$	89 files	0.363s	0.447s	$\approx$ 23%
171 $MBytes$	1108 files	14.644s	18.068s	$\approx$ 23%

The results presented in Table 3 were obtained by decompressing packed archives into the local file system and into the FEW Phone FS, using the 'tar -xzf' command on different size packets. This experiment evaluates the performance of write operations.

Table 3. Measured times for extracting files from an archive.

Archive Size	**Number of Files**	**Unpacked Size**	Extracting files to		**Overhead**
			Local FS	**FEW Phone FS**	
100 $kBytes$	84 files	648 $kBytes$	0.082s	0.243s	$\approx$ 296%
495 $kBytes$	89 files	2.3 $MBytes$	0.110s	0.367s	$\approx$ 334%
151 $MBytes$	1108 files	171 $MBytes$	16.876s	25.590s	$\approx$ 151%

These results present a non-negligible overhead when compared to the values obtained from the extraction of the same contents to the local file system. This overhead is due to the management of information for reconciliation use, which is inefficient in our current implementation. Although the overhead is large, we believe that for common uses, this does not affect the usability of the system, as it is hardly noticeable for the user - for example, when writing 89 files with a total size of 495KB, the total execution time is still under 0,5 seconds.

4.2 Reconciliation Evaluation

The following results were obtained during the reconciliation process. The results presented in Table 4 were obtained during the reconciliation of two already synchronized nodes. In these results it is possible to see the low overhead of this process, thus showing the advantages of the update detection methods used.

Table 4. Measured reconciliation times for synchronized nodes.

PC to iPod using Wi-Fi
$0.396s$

The results presented in Table 5 were obtained during the reconciliation process, using two different set sizes, with a clean server side cache (first reconciliation). From these results it is possible to see the limitations imposed by the low bandwidth of Bluetooth technology.

Table 5. Measured first reconciliation times.

Number of Files	**Total Size**	**Reconciliation Time**		
		PC to PC		**PC to iPod**
		Bluetooth	Wi-Fi	Wi-Fi
84 files	648 $kBytes$	$17.276s$	$6.876s$	$35.762s$
89 files	2.3 $MBytes$	$30.158s$	$8.856s$	$51.706s$

The results presented in Table 6 were obtained during the reconciliation of a second PC with the iPod Touch, after the initial synchronization. These results present the improvement in performance when using the information retrieved by the DSV module. Using peer-to-peer interaction, not only improves system performance, but also reduces the volume of data transferred from the server.

Table 6. Measured second reconciliation times.

Number of Files	**Total Size**	**Reconciliation Time**		**Gain**
		with Peer-to-Peer	without Peer-to-Peer	
84 files	648 $kBytes$	$14.877s$	$16.921s$	$\approx 14\%$
89 files	2.3 $MBytes$	$26.338s$	$30.393s$	$\approx 15\%$

The results obtained during the reconciliation of multimedia contents are presented in Table 7. These results were obtained running the server with a clean cache, and with a client storing two 15 mega pixels digital photographs, with

approximately 4MBytes each. The two photos were transcoded to a resolution of 800x533, that resulted in a reduction in size to approximately 33kBytes.

Table 7. Measured multimedia reconciliation times.

Number of Files	**Total Size**	**Reconciliation Time**		**Gain**
		with Transcoding	without Transcoding	
2 files	8 $MBytes$	8.549s	27.761s	$\approx$ 325%

From the obtained evaluation results, it is possible to conclude that the use of both the Data Transcoding and the Data Source Verification modules has proved to be beneficial for the performance of the system, improving the system's performance, and also reducing communications and storage overhead.

5 Related Work

A large number of distributed file systems have been implemented (e.g. Ficus [5], Coda [1]), allowing users to remotely access their files. Some of these systems can be used in mobile computing environments and include support for disconnected operation. However, the complexity associated with setting up a new server and using it in a network environment with private networks and firewalls lead most users to prefer using portable storage devices to transport their data.

Lookaside caching's [9] allows for portable storage devices (PSD) to be used as availability extensions and performance enhancers for distributed file systems (DFS). Although Lookaside caching allows for disconnected operations, users can only access a small portion of their data during disconnection, as the portable storage device is used as a cache. Additionally, this approach still requires the use of a distributed file server, incurring in the same drawbacks of distributed file systems.

PersonalRAID [10] allows a PSD to be used for propagating updates among several personal replicas. However, as the system only maintains updates in PSDs, makes it impossible for a user to access all his data in a new computer.

Footloose [11] introduces the concept of physical eventual consistency. This concept allows for the mobile phone to be used as the means to propagate updates between disconnected nodes. This is possible since the mobile phone stores the most recent version of the user's data, and it is "carried around" by the user. FEW Phone FS extends the approach of Footloose by allowing multimedia data to be transported efficiently and by allowing clients to obtain data contents from other replicas (even outside the system), thus minimizing the requirements of the mobile phone.

Like Coda [1], the FEW Phone FS also uses a log structure for maintaining directory updates. Although both systems automatically deal with directory update conflicts, the solution used by Coda relies greatly on servers. This solution

is not suitable for the FEW Phone FS due to the limitations of mobile devices. Coda deals with file update conflicts using application specific resolvers. Other system, like Pangaea [3] also use this approach. This approach could also be used in the FEW Phone FS.

EnsemBlue [12] is a distributed file system for personal multimedia that incorporates both general-purpose computers and consumer electronic devices. The system transcodes data, based on application needs, using what the authors describe as 'persistent queries'. In the FEW Phone FS the Data Transcoding module is used during the reconciliation process for adapting multimedia contents based on the specifications of the mobile device.

quFiles [13] provides a mechanism to support multiple fidelities of data within the same file system. The transcoding process, in a normal scenario, only happens once for each file. quFiles transcodes files when they are stored in the system, maintaining multiple transcoded versions of each file. The FEW Phone FS transcodes files "on-the-fly" during the first synchronization with the mobile phone, since these are transcoded according to the specific device's specifications.

6 Conclusions

The main goal of this work was to design and develop a distributed data management system, that would allow users to access their personal data in any machine, independently of location and network connectivity.

For this purpose, the FEW Phone File System was designed to take advantage of current mobile phones' storage and wireless communication capabilities for maintaining the most up-to-date version of the users data, and allowing for replicas to be created and accessed whenever needed.

The combination of the client/server model with peer-to-peer interaction leads to an hybrid architecture. Relying on the mobile phone as a portable server allows the system to use it to provide data availability at any time. Peer-to-peer interaction allows for power consumption to be reduced and for improving the system's performance, as the presented results have shown.

The Data Transcoding module allows to reduce multimedia data transferred to and stored on the mobile phone, without compromising quality when these files are accessed in the mobile phone. This can be interesting for example, for digital photographs that a user wants to keep in both the mobile phone and his desktop computers.

The Data Source Verification module allows the system to record the sources for the files stored in the system, including remote sources (not in the system). This approach explores the common case where files stored were obtained from the Web or are stored in some remote server (e.g. CVS/SVN). This module also stores the location of other replicas in the system. This information is used by clients to obtain replica contents from other clients. To our knowledge, the FEW Phone File System is the first system to use this approach to improve performance and availability. Moreover, this also extends battery life by reducing communications with the mobile device, while improving performance.

By combining the Data Source Verification module with the Data Transcoding module, the system allows clients to always access full-fidelity contents. This is a new feature when compared with previous solutions that used data transcoding.

The results of the performance tests showed that the proposed design imposes minimal overhead to data access on the clients, after initial synchronization. Also, the use of the Data Transcoding and Data Source Verification modules allow the FEW Phone File System to achieve its goals.

References

[1] Braam, P.J.: The Coda Distributed File System. Linux J. (1998)

[2] Parker, D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of Mutual Inconsistency in Distributed Systems. IEEE Trans. Softw. Eng. **9**(3) (1983) 240–247

[3] Saito, Y., Karamanolis, C., Karlsson, M., Mahalingam, M.: Taming aggressive replication in the Pangaea wide-area file system. In: Proc. OSDI. (2002) 15–30

[4] Guy, R.G., Reiher, P.L., Ratner, D., Gunter, M., Ma, W., Popek, G.J.: Rumor: Mobile Data Access Through Optimistic Peer-to-Peer Replication. In: ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining, London, UK, Springer-Verlag (1999) 254–265

[5] Guy, R.G., Heidemann, J.S., Mak, W., Popek, G.J., Rothmeier, D.: Implementation of the Ficus Replicated File System. In: USENIX Conference Proceedings. (1990) 63–71

[6] João Soares: FEW Phone File System. Master's thesis, Faculdade de Ciências e Tecnologia, Portugal (April 2009)

[7] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565

[8] Henk, C., Szeredi, M., Pavlinusic, D., Dawe, R., Delafond, S.: Filesystem in Userspace (FUSE) (December 2008) http://fuse.sourceforge.net/.

[9] Tolia, N., Kozuch, M., Satyanarayanan, M.: Integrating portable and distributed storage. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies. (2004) 227–238

[10] Sobti, S., Garg, N., Zhang, C., Yu, X., R, A.K., Wang, O.Y.: PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In: Proc. First Conference on File and Storage Technologies. (2002) 159–174

[11] Paluska, J., Saff, D., Yeh, T., Chen, K.: Footloose: a case for physical eventual consistency and selective conflict resolution. Mobile Computing Systems and Applications, 2003. Proceedings. Fifth IEEE Workshop on (Oct. 2003) 170–179

[12] Peek, D., Flinn, J.: EnsemBlue: Integrating distributed storage and consumer electronics. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. ACM SIGOPS. (2006) 219–232

[13] Veeraraghavan, K., Nightingale, E.B., Flinn, J., Noble, B.: quFiles: a unifying abstraction for mobile data management. In: HotMobile '08: Proceedings of the 9th workshop on Mobile computing systems and applications, New York, NY, USA, ACM (2008) 65–68