quFiles: A Unifying Abstraction for Mobile Data Management

Kaushik Veeraraghavan*, Edmund B. Nightingale †, Jason Flinn* and Brian Noble*
University of Michigan* Microsoft Research (Redmond)†

ABSTRACT

We introduce a unifying file system abstraction, called a quFile, that provides a new mechanism for implementing mobile data management policies. quFiles allow arbitrary data types to be bundled together without confusing the user. Similar to a quBit (quantum bit), the particular data displayed by a quFile is not determined until the moment it is observed. A quFile displays the appropriate data type and version depending upon an application-specific policy that can take any information into account, such as the platform, external devices, context, connectivity, or battery power. We first describe what the quFile mechanism provides to applications and mobile devices. We then discuss how quFiles can benefit mobile data management in the areas of resource management, extensibility, and data consistency and availability.

1. INTRODUCTION

Today, no single abstraction exists that enables users and applications to address all the problems inherent in data access in a mobile environment. This is no coincidence. To systems designers, application developers and end-users alike, mobile computing still presents a harsh and unforgiving environment. Applications must contend with hard energy constraints, highly variable connectivity, bandwidth and latency limitations, computing contexts that alternate between trusted and untrusted environments, and a large, diverse set of platforms that each present unique constraints on the format and fidelity of data.

In the past, new policies that mitigated these problems required new software systems, each of which implemented narrow, problem-specific solutions [15, 5, 1]. The result is a collection of ad-hoc implementations and abstractions for managing data in a mobile environment.

This complex world calls for sophisticated, application and platform specific policies for data management. Yet, after twenty years of mobile computing, managing data in a mobile environment is still an experiment in frustration: despite large amounts of data shared among devices, manual or application-specific management is still the current state-of-the-art. Most applications fail to adapt as connectivity changes, and many mobile devices fail to do more than dim the screen as the battery is depleted. Finally, as the behavior of the user changes, mobile systems fail to take notice.

To simplify mobile data access and management, we introduce quFiles, which provide a single abstraction for implementing solutions to disparate problems such as extensibility or power management. Similar to a quBit (quantum bit), the particular data displayed by a quFile is not determined until the moment it is needed. A quFile displays the appropriate data type and version depending upon an application specific policy that can take any information into account, such as the screen size, platform, external devices, context, connectivity and battery power.

Support within the file system for application-specific policies bridges the semantic gap without modifying applications. Anyone can write policies that extend applications to work with quFiles. Multiple policies (e.g., power and availability) may be strung together to decide which data an application requires.

We are currently implementing a prototype of quFiles in BlueFS [14], a distributed file system with first-class support for battery powered, mobile devices such as laptops and cell phones. While our current implementation is limited to BlueFS, we believe that the design of quFiles described in Section 2 is general enough to allow most local and distributed file systems to add the quFile abstraction. Section 3 delves into the uses of quFiles in a wide variety of problem domains, including device/platform specific resource management, extensibility, data consistency and availability. We present related work in Section 4 and then conclude.

2. WHAT IS A QUFILE?

Simply put, a quFile encapsulates different representations of a single data object. For example, a quFile that encapsulates musical data could have several different formats that are each an encoding of the same musical data (e.g., an mp3 file, an m4a file and a WAV file). A quFile dynamically resolves to zero or one of its encapsulated representations depending on the context in which it is accessed. Examples of such context are: the application that is accessing the

data, the mobile device on which the content is accessed, the battery state of the mobile device, and the network connectivity of the device.

Our design of quFiles attempts to balance several important goals:

- **Transparency:** Ideally, the existence of a quFile should be completely hidden from the user. The complexity of multiple formats or representations of data should be encapsulated in the file system.
- **Backward compatibility:** quFiles should work with existing applications.
- **Extensibility:** quFiles should support arbitrary resolution policies. This implies that policies should be specified via code that is safely and dynamically linked into the file system.
- **Simplicity:** Enhancing an existing file system to use quFiles should require a minimal set of changes.

In order to realize our first goal of transparency, the creation and maintenance of quFiles is done automatically. When a user adds an audio file, e.g. song.mp3, to the BlueFS namespace, it is copied to the server and a file system change notification is generated. In our BlueFS implementation, a persistent query [16] is used to deliver this notification to a transcoding application on our file system server. On a device with a local file system, the transcoding application will only run if the device is resource-rich i.e., connected to A/C power and has spare storage capacity.

The transcoding application creates a quFile and populates it with the original file (song.mp3), as well as additional formats such as song.wav and song.m4a. The user can modify the quFile using the original file name. If the user later deletes song.mp3, the entire quFile, including the original file and the additional formats, is deleted. If the user moves song.mp3 to a new directory, the entire quFile is moved to that directory. To make this process as transparent as possible, the resolution policy for a quFile should return the original file when the invoking application is a utility such as `ls` or a graphical file system browser.

By default, we do not permit the user to edit the low-fidelity transcoded copies of a file unless the transcoding algorithm is lossless so that the transformation loses no data. Additionally, we envision that applications might provide custom policies that keep a log of the operations performed by the edit — these operations can be replayed on the high-fidelity original to ensure that no data is lost. Finally, applications can also provide custom policies that parse edits and apply them across files — for instance, a policy can check if any writes to audio files occur within an ID3 tag and if so, permit the edit and repeat it on the original file.

In order to realize our third goal of extensibility, our BlueFS implementation uses Sprockets [17], which use a form of software fault isolation to allow the file system to safely execute dynamically specified code modules. Each quFile contains a link to a shared library that specifies its particular resolution policy; links are used to avoid the storage overhead caused by duplicating the same resolution policy across many quFiles. Thus, much like an object in an object-oriented programming language, each quFile encapsulates both data and the mechanisms for accessing that data.

We accomplish our final goal of simplicity by noting that the on-disk representation of a quFile is quite similar to that of a directory. Both directories and quFiles contain a set of names and references to other objects in the file system. While the semantics of the two objects differ significantly, we can reuse a substantial portion of the original file system code by implementing quFiles as directories that either have a reserved name (our current implementation treats all directories with the name .qufile.<name> as a quFile) or a reserved number in the file mode portion of the metadata.

The file system implementation need only interpose on a small subset of directory operations. On a `readdir` operation, the file system should invoke the resolver for each quFile contained within the directory to determine which representation contained in the quFile will be returned to the application. This name, rather than the name of the quFile, is specified in the data returned by `readdir`. Similarly, on `lookup`, the file system must return the object contained within the quFile rather than the quFile itself. Other operations, such as `create` and `mkdir` should guarantee uniqueness across all regular objects within the directory, as well as all representations contained by quFiles within that directory. With these exceptions, a file system can reuse its existing directory code to support quFiles.

3. QUICK AND EASY CONSTRUCTS WITH QUFILES

We currently envision quFiles benefiting mobile clients in three areas: resource management, extensibility and consistency/availability.

3.1 Resource management

Resource management on a mobile device is a complex problem because all available resources are severely constrained. Each decision affecting the CPU, network bandwidth, storage or power usually affects other resources in the system. Further, the semantic gap has frustrated system builders: the operating system has global knowledge of available resources, and only the application knows its intent for resource consumption. In the past, researchers have managed these problems in two ways: first, the problem is constrained by considering only a subset of the problem, such as power or availability. Second, the semantic gap is bridged either by exposing more information to the application (requiring each application to make an independent decision), or the application is modified to reveal its intent to the operating system. Each new policy has required a new system to be built (e.g., Odyssey [15] or STPM [1]), which makes putting policies from different problem areas together virtually impossible. quFiles simplify the problem of resource management by allowing applications to reveal their intent without modifying applications or the operating system.

quFiles can extend the battery lifetime of a mobile device by taking advantage of its spare storage capacity. Traditionally, lossy compression has been used on mobile devices because storage space was precious and network bandwidth was constrained; an mp3 transfers faster and takes up less space than its lossless (WAV) counterpart. However, a compressed file takes significantly more computational power to decode than the original, lossless version. Our preliminary measurements show that playing a WAV file rather than an mp3 increases the battery lifetime of an iPAQ by 11%.

As described in Section 2, when an mp3 is added to the file system, a transcoder generates its corresponding WAV and m4a versions and moves the original and transcoded files to a newly created quFile. When a mobile device reads the quFile, an application-specific policy determines which audio format to return.

One sample policy could function as follows. If the mobile device is connected to the BlueFS server and is running off A/C power, the application's energy-conservation policy would dictate that apart from the mp3 being played, the power-efficient WAV version of the audio file should also be cached on the device. quFiles leverage the caching mechanism built into BlueFS to fetch the WAV transcoding from the server and cache it on the device. The next time the user attempts to play this audio file, the quFile will return the energy-efficient WAV file. If only one version is available, the policy will determine if it is more efficient to decode the version in local storage or fetch an alternate version from the file server.

On some mobile devices, storage is constrained. When storage becomes scarce, application-specific policies can be invoked to determine which files can be safely deleted. For backup purposes, only the original file need be retained since alternate versions can be transcoded from the original.

Format replication, where a file might be encoded in multiple formats to cater to application or device constraints, also means that a device might cache a file type that it is incapable of playing (e.g., some cell phones cannot play audio files encoded in an m4a format intended for an iPod). quFiles absolve the user from having to deal with format replication; device-specific policies can direct each device to only cache formats that they are capable of playing.

3.2 Extensibility

In addition to resource management, quFiles allow us to easily support various extensions of data objects. For example, *context* has become increasingly important in mobile and pervasive systems. Put simply, an object's context is a collection of metadata elements describing various faces of the object—where it was obtained, who else collaborated on it, what it is related to, and so on. By supporting metadata decoration, such as a tuple space [9], quFiles provide a simple abstraction for applications that harvest, reason about, and share context. By packing them together, applications and services that are ignorant of context can still treat the collection as a unit. This preserves contextual information for interested applications and services.

One can also apply this notion of extensibility to protect against format obsolescence. For example, applications such as Corel Draw (a photo editing suite) and Lotus123 (an office application suite) once were popular, if not dominant, but have since fallen out of favor. As such applications die out, their file formats become increasingly difficult to manage—over time, format conversion tools are harder to come by. However, for a time, these file formats were just as popular as their eventually successful competitors. During that time, conversion tools are plentiful. quFiles can exploit such transitional periods by converting between common formats, creating format-specific conversion policies out of existing conversion tools. Even when formats are popular and current, some individual clients may not be capable of reading them using only the installed base of software—quFiles also support such machines by pre-converting to formats that are known to be used by that client. While this is not a problem specific to mobile and pervasive computing, such platforms exacerbate the problem, due to their highly constrained resources and often unique display capabilities.

3.3 Consistency and Availability

Format replication compounds the problem of ensuring the most up to date version of a file that exists on multiple mobile devices is available when the user requires it. Distributed file systems have handled this problem [12] by either forcing the user to fix the conflict, or by building application-specific resolvers. quFiles allow application developers to quickly and easily add structure to application data or metadata that allows conflicting versions of a file to be resolved automatically.

Distributed file systems have typically made no visible distinction between data cached locally and data that must be fetched remotely from a server. Exposing this information to the user is not necessarily useful, but quFiles can determine dynamically whether to show any file at all. For example, one useful policy for multimedia data is to not show files if they must be fetched from a remote server and the available bandwidth is less than the bit-rate of the multimedia file. Using this policy, a music or video player will only see content that it can play at the moment. Alternatively, if battery lifetime is of utmost importance, a quFile policy might not make available files that must be fetched over the network.

4. RELATED WORK

We believe quFiles are a novel abstraction for unifying disparate mobile data management policies. Our design describes how quFiles can be easily incorporated into existing local and distributed file systems. Application developers can specify how quFiles should resolve custom data-types by specifying device or context-dependent policies.

There are many possible ways to implement the quFile abstraction. We chose to use Sprockets [17] as a method to safely and dynamically link type specific code into the file system. Alternatively, we could have used methods such as Watchdogs [3] or the FUSE toolkit [8] that allow user level file system extensions. Similarly, we chose to use persistent queries [16] to notify transcoders that quFiles need to be created or updated. Operating system specific notification

mechanisms such as the NTFS change log [4] or Linux's `inotify` [13] could also serve this purpose.

quFiles provide a mechanism to support multiple fidelities of data within the file system. Other alternatives such as application-aware adaptation [15], component-based adaptation [5] or NTFS data streams [18] require either application modification or the insertion of a proxy between the application and the file system.

Schilit et. al. introduced context-aware computing computing applications [19] and identify four major categories of applications. Of these, quFiles support both contextual information and command-based applications, and context-triggered actions. While Schilit focused on usability and the GUI used to interact with the system, quFiles adapt to the application and usage context and return different views of the user's file system. This adaptation allows quFiles to resolve to the correct representation for every scenario, including ones that are counter intuitive. For instance, Barr and Asanovic [2] argue that data compression before transmission greatly reduces the energy expended by a mobile device. We note that the opposite holds true for local storage — uncompressed data requires less energy to play than compressed data, hence compression before transmission might not always be the best policy.

Past approaches such as Xerox's Placeless Documents [6] and Gifford's Semantic File Systems [10] suggest semantic or property-based mechanisms as alternatives to organize data in a file system. quFiles share the same goal with Placeless Documents and Semantic File Systems, but we have chosen a backward-compatible design that works within existing file systems, rather than requiring a system re-write.

Fox [7] and Gribble [11] introduce the notion of an intermediary "active proxy", that adapts data flowing from resource-rich servers to thin devices. An active proxy is trusted by the device to perform the adaptation correctly on its behalf. In comparison, quFiles offer a mechanism whereby applications specify custom policies that dictate how data can be transformed based on power, network connectivity and similar context. quFile policies execute within the file system address space with guards enforcing file system integrity [17].

5. CONCLUSION

We have shown that quFiles can provide a unifying abstraction for implementing previously disparate mobile data management policies. Some uses of quFiles include resource management, extensibility and data availability. We plan to extend both local and distributed file systems to support quFiles. Using this support, we will implement novel policies such as using spare storage capacity to extend battery lifetime. We will also demonstrate the generality of quFiles by implementing policies that were previously realized by problem-specific solutions. We hope that quFiles will prove to be an abstraction that greatly simplifies mobile data management.

6. ACKNOWLEDGMENTS

We thank Dan Peek, Ya-Yunn Su, Mona Attariyan, Benji Wester, Jon Oberheide, Manish Anand and the anonymous reviewers for suggestions that improved the quality of this paper. The work is supported by the National Science Foundation under awards CNS-0306251 and CNS-0509089. Jason Flinn is supported by NSF CAREER award CNS-0346686. Intel Corp. has provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, the University of Michigan, or the U.S. government

7. REFERENCES

[1] Anand, M., Nightingale, E. B., and Flinn, J. Self-tuning wireless network power management. In *MobiCom* (2003), pp. 176–189.

[2] Barr, K., and Asanović, K. Energy aware lossless data compression. In *MobiSys* (2003), pp. 231–244.

[3] Bershad, B. B., and Pinkerton, C. B. Watchdogs - extending the unix file system. *Computer Systems 1*, 2 (1988).

[4] Cooperstein, J., and Richter, J. Keeping an eye on your NTFS drives: the Windows 2000 Change Journal explained. *Microsft Systems Journal* (1999).

[5] de Lara, E., Wallach, D. S., and Zwaenepoel, W. Puppeteer: Component-based adaptation for mobile computing. In *USITS* (2001).

[6] Dourish, P., Edwards, W. K., LaMarca, A., Lamping, J., Petersen, K., Salisbury, M., Terry, D. B., and Thornton, J. Extending document management systems with user-specific active properties. *ACM TOIS 18*, 2 (2000), 140–170.

[7] Fox, A., Gribble, S. D., Brewer, E. A., and Amir, E. Adapting to network and client variability via on-demand dynamic distillation. In *ASPLOS* (1996), pp. 160–170.

[8] Filesystem in userspace. http://fuse.sourceforge.net/.

[9] Gelernter, D. Generative communication in Linda. *ACM TOPLAS 7*, 1 (1985).

[10] Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole, J. W. Semantic file systems. In *SOSP* (1991), pp. 16–25.

[11] Gribble, S. D., Welsh, M., von Behren, J. R., Brewer, E. A., Culler, D. E., Borisov, N., Czerwinski, S. E., Gummadi, R., Hill, J. R., Joseph, A. D., Katz, R. H., Mao, Z. M., Ross, S., and Zhao, B. Y. The ninja architecture for robust internet-scale systems and services. *Computer Networks 35*, 4 (2001), 473–497.

[12] Kumar, P., and Satyanarayanan, M. Flexible and safe resolution of file conflicts. In *USENIX Winter Technical* (1995).

[13] Love, R. Kernel korner: Intro to inotify. *Linux Journal*, 139 (2005), 8.

[14] Nightingale, E. B., and Flinn, J. Energy-efficiency and storage flexibility in the Blue File System. In *OSDI* (2004), pp. 363–378.

[15] Noble, B. D., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J., and Walker, K. R. Agile application-aware adaptation for mobility. In *SOSP* (1997), pp. 276–287.

[16] Peek, D., and Flinn, J. EnsemBlue: Integrating consumer electronics and distributed storage. In *OSDI* (2006), pp. 219–232.

[17] Peek, D., Nightingale, E. B., Higgins, B. D., Kumar, P., and Flinn, J. Sprockets: Safe extensions for distributed file systems. In *USENIX Annual Technical* (2007), pp. 115–128.

[18] Russinovich, M. E., and Solomon, D. A. Advanced features of NTFS. In *Microsoft Windows Internals* (2005), Microsoft Press, pp. 719–721.

[19] Schilit, B., Adams, N., and Want, R. Context-aware computing applications. In *IEEE WMCSA* (1994).