

◆ Augmenting Handset Capacity Through Virtual Storage

Supratim Deb, Ankur Jain, Anirban Majumder, K. V. M. Naidu, Jeyashankher Ramamirtham, Rajeev Rastogi, and Anand Srinivasan

Current cell phones are capable of accessing the Internet, downloading music and games, taking pictures, and recording videos. These activities can easily result in a huge amount of multimedia content requiring tens or hundreds of gigabytes of storage, which is several orders of magnitude more than the storage capacity of today's cell phones. In this paper, we propose a novel remote storage solution to increase the amount of storage accessible to a mobile user. Our solution consists of two components: a client program running on the cell phone and a remote server to manage remote storage. The novelty of our solution results from the client program that makes the accesses to the remote storage completely transparent and seamless to the user. Thus, the user's perception of the amount of storage available on the cell phone is considerably higher than the actual capacity of the phone. In addition, the client program implements sophisticated caching algorithms and wireless optimizations to ensure that the user-perceived performance is not significantly impacted. © 2007 Alcatel-Lucent.

Introduction

Storage requirements are expected to grow significantly in the coming years as more end users opt to store a large number of songs, videos, games, photographs, documents, PowerPoint slides, and other media in their handsets. End users would like to have access to their large-capacity storage from anywhere, even while they are away from their personal computers or laptops. A convenient way for an end user to access this information is to use his cell phone. However, providing substantial storage on the phone itself is not possible because of its small form factor and the expensive nature of very small storage devices (such as Flash disks and 1-inch hard disks). Hence, a user needs to be able conveniently to access data that

are stored elsewhere, that is, on a remote server or home PC or laptop. We present a novel way to store and access data remotely such that the access to the remote data is completely seamless and transparent to the end user as well as to the user-level applications that run on the phone.

Today, a user with a cell phone that allows Web access can utilize online Web-based storage repositories. These storage repositories allow a user to store all the data on a Web site and then access them anywhere using a Web-enabled device. In particular, a user can log on to the storage Web site from a Web-enabled phone and then have access to all the files that he/she had uploaded onto that Web site. There are several

Panel 1. Abbreviations, Acronyms, and Terms

API—Application programming interface
BREW—Binary runtime environment for wireless
DD—Download directory
DF—Download file
J2ME—Java 2 Platform, Micro Edition
HTTP—Hypertext Transfer Protocol
I/O—Input/output
IP—Internet Protocol
KB—Kilobyte

LFU—Least frequently used
MB—Megabyte
NetStorM—Network Storage for Mobiles
PC—Personal computer
RAID—Redundant array of independent disks
TCP—Transmission Control Protocol
UF—Upload file
UI—User interface
WebTV—Web television

disadvantages of this approach. First, performing the activities of going to the Web portal, logging in, and downloading files is extremely inconvenient because of the small form factor of the phone. Second, deleting the appropriate set of files on the phone to make space for new files that need to be downloaded is also a nontrivial, time-consuming, and inconvenient task. In other words, the remote storage is not transparent to the user and this is one of the chief obstacles for a user to use remote Web-based storage instead of local storage devices. Third, the performance of remote storage devices is significantly poorer than performance experienced with local storage, because accessing Web-based storage is not optimized for access over wireless. Specifically, Web requests are made using Hypertext Transfer Protocol (HTTP) that runs over Transmission Control Protocol (TCP), which is known to perform poorly in high error rate wireless environments. Furthermore, HTTP is wireless connection agnostic and it was originally designed for reliable wireline links with stable connection quality. In contrast, wireless link quality can vary considerably, depending on user location and the number of other users sharing the wireless connection. Last but not least, Web-based storage systems do not cache content on the local device—as a result, every request needs to be satisfied remotely, potentially leading to a significant delay in the download of each file.

We propose a virtual network storage-based solution to the problem. Our solution has two main components: (1) a client program running on the handset and (2) a server running remotely to handle the remote storage. In the case of Windows*-based systems, the

client program consists of a file system filter driver, while in Symbian*- or Linux*-based systems, the client is a file system hook. In all these implementations, the client program captures all the file system calls, and depending on the call, it performs upload/download of data to/from the remote storage device in order to make user applications behave as if the data were from local storage. Because data management is automated, as an additional benefit, our solution results in automatic backup of the user's data. Installing a redundant array of independent disks (RAID) on the remote storage server makes the storage more reliable and resilient to handset loss and theft. For phones that do not run Windows, Symbian, or Linux, but support the Binary Runtime for Wireless (BREW*) or the Java* 2 Platform Micro Edition (J2ME*) platforms, we propose another variant of the solution. For such systems, we provide an application that makes it easier for the user to access the remote storage. Though this application does not have the benefit of transparency, we implement sophisticated caching and streaming algorithms to improve storage performance. Finally, we note that our solution can be viewed as a mobile handset-optimized network file system. While there has been other work on network file systems in the low-bandwidth environment [1–5], none of these studies is tailored to the requirements of a mobile handset.

We now describe our solution in more detail. We first discuss how our solution can be applied to handsets with an advanced operating system like Windows CE and later discuss our solution for handsets running BREW or J2ME without any advanced operating systems.

Implementation for Handsets With Advanced Operating Systems

We now describe the primary variant of our solution that is applicable to handsets running Windows CE. (The solution for handsets running Symbian or Linux is similar.)

Architecture

A schematic representation of the system architecture is shown in **Figure 1**. The main component of our solution resides in the client program on the mobile phone and consists of the following blocks:

- *File system call classifier*. The role of this component, installed above the driver, is to check which file system calls should be handled by the driver. The file system calls that are not to be handled by the driver are passed directly on to the

underlying file system. The rest of the calls are passed on to the driver. In general, the implementation can be applied to a specific directory, which we henceforth refer to as the NetStorM directory. The file system call classifier passes all file system calls (e.g., create, read, write) for all legitimate user files to the driver. Finally, there could be other files in the NetStorM directory that are not created by the user but that are required by our solution; our driver ensures that these files are not visible to user applications and blocks all the file system calls for these files.

- *File downloader/uploader*. The file downloader/uploader interacts with the driver and the remote storage server. This component is responsible for getting entire files or relevant portions of files

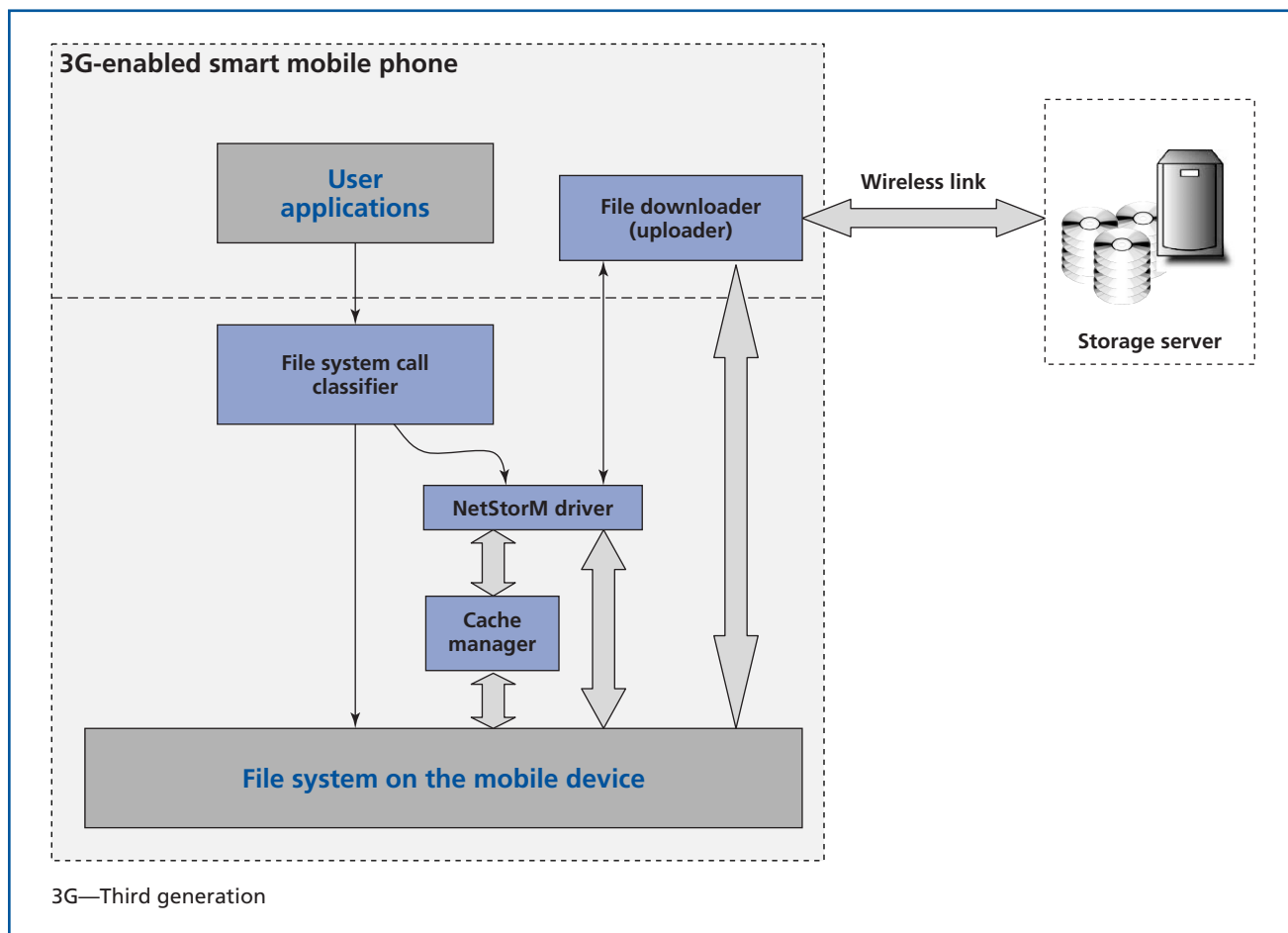


Figure 1.
Block diagram of system.

from the remote storage server upon receiving requests from the driver. Furthermore, this component also uploads files from the mobile device to the remote storage device on request from the cache manager.

- *Cache manager.* The cache manager decides how much of which file to store locally in the mobile device. There can also be a sophisticated statistic collector in the cache manager that tracks user access patterns to determine the locally stored files. We discuss the design of the cache manager in more detail in a later section.
- *Driver.* The driver is the main component of our solution. All the file system calls for the NetStorM directory go to the driver before reaching the file system. On receiving a file system call, the driver first interacts with the cache manager to check whether or not the file is locally present in the mobile device. If the file is locally present, then the file system call is passed on to the underlying file system. On the other hand, if the file is not locally present, then the driver's actions depend on the file system call. For example, if there is a read call for a file that is not locally present in the mobile device, the driver sends a request to the file downloader to download the required file. The details of how the driver handles the different file system calls are provided in the next section.

Handling I/O Calls

The primary goal of the driver is to make accesses to the remote storage device completely seamless and transparent to the user. This transparency is achieved in the following manner:

1. When a user tries to access a file that is already resident in local storage, the driver passes on all related file system calls to the underlying file system without any modification.
2. When a user tries to access a file that is not present in local storage, the driver first determines whether there is sufficient space for the file in local storage. If so, it goes ahead and starts downloading the file from the remote storage device. Otherwise, it identifies certain files for deletion, deletes them, and then starts the download. All these activities are performed in a separate

prefetching thread, which is started when an "open" command is invoked on the file. This allows the "open" call on the file to complete quickly, without waiting for the file to be completely downloaded. The "read," "write," and other file system calls are handled in the following manner:

- When a "read" is invoked on the file, the driver first waits for the prefetching thread to complete the download of the file or at least complete the download of the relevant portion of the file. It then returns the relevant bytes to the user application that invoked the "read" call.
 - A "write" call is easier to handle than a "read" call because the driver does not need to wait for the download to complete. It just needs to record the offset and the number of bytes involved in the write operation.
 - On receiving a file listing call for a directory, the driver first checks the local availability of a special file called *dirinfo* corresponding to that particular directory. If the *dirinfo* file for that directory is not available locally, then it is downloaded from the remote folder. To obtain better performance, the download of the appropriate *dirinfo* file can be started when a user application tries to open a handle to the corresponding directory. The *dirinfo* file contains the metadata information about the files in that directory, including the file names and the various file attributes. The driver simply opens the appropriate *dirinfo* file and returns the file listing of the requested directory.
 - When a user application tries to obtain (or set) the attributes of a file including the file size, the driver opens the parent directory's *dirinfo* and returns (or changes, respectively) the appropriate values.
 - All the other file system calls are either handled in a similar fashion or directly passed on to the file system.
3. In addition to the preceding operations, the driver regularly uploads modified files to the remote storage server. Uploads are performed in the background, and at a lower priority, to minimize the

impact on user-perceived performance (unless the file is being deleted, in which case the upload thread is temporarily assigned a higher priority). In addition, uploading can be postponed if the battery level is low to conserve the battery for regular phone usage. Similarly, if the connection speed is low because of poor link quality, then the upload can be deferred to a later time or carried out at a slower rate. Further, depending on the signal quality and battery level, we decide whether to utilize compression for uploads. In particular, if the battery level is high and signal quality is low, then compression is used; on the other hand, if the battery level and signal quality are both high, then compression is not used. Note that the client software on the handset is responsible for collecting the statistics on signal quality, connection speed, application response times, and other parameters. This is done by making the appropriate operating system calls. (More detailed information can be achieved by snooping on IP packets sent/received by the handset.)

Observe that point 1 implies that the files that are cached in local storage can be accessed very quickly. Thus, the caching algorithm, which determines the files that are kept in local storage, is a very important indicator of user-perceived performance. Additionally, point 2 mentions that a “read” call needs to be delayed only until the relevant bytes are downloaded into local storage. For audio and video files, this is equivalent to streaming the content instead of waiting for the full content to be downloaded before playing it. Essentially, audio/video playback is begun once a small initial portion of the clip has been downloaded, and the remaining portion is downloaded concurrently with playback. This considerably decreases the user-perceived latency.

Buffering Policy for Read Calls

Before a read call is passed to the file system, we need to ensure that the required portion of the file is already downloaded. Implementing this in an ad hoc fashion can lead to a bad user experience when playing a video: the video may play and stop intermittently if the rate of reading the file is higher than the network download speed. This should be prevented and, hence, there is a need to make the read call wait.

The waiting period must be chosen appropriately so as to minimize interruptions once the video starts playing. Given the number of acceptable interruptions, we determine the waiting time for a read call on the basis of the current available bandwidth and the size of the file. The key idea is to delay the read calls only when the read-offset exceeds certain “checkpoints,” which are calculated on the basis of the file size and the number of acceptable interruptions. The waiting time is then computed in a way to ensure that if a read call is allowed to go through, then all the bytes until the next checkpoint can be read without any further wait. Note that the download rate also needs to be taken into account since there is no need to delay a read call if the download rate is higher than the read rate.

Implementation on the J2ME and BREW Platforms

We have also developed user-level applications for handsets that support platforms like BREW or J2ME but do not run an advanced operating system. For such systems, we provide an application that makes it easier for the user to access the remote storage device. Though this application does not provide complete transparency, it does implement our sophisticated caching algorithms and other enhancements that are described in the succeeding sections. This results in performance that is superior to traditional Web-based storage. Basically, our application provides users with an interface for browsing files in remote storage as well as for uploading new files.

This implementation consists of a user interface component and a client for downloading and uploading files, as well as file metadata information. We now describe these in more detail.

User Interface

The user interface (UI) begins with a listing of the root directory on the remote storage device: that is, the screen shows a scrollable list of text items containing the names of the files and subdirectories in the root directory. The user can use the standard arrow keys to navigate the directory. The directory listing includes a link to the parent directory in the form of “. .”.

A user can select one of the items (file or subdirectory) listed using the select key on the phone. This results in the following behavior:

- Selecting a subdirectory allows the user to navigate inside that directory. This requires the UI component to display the file and subdirectory listing for that selected directory. As in the Windows CE implementation, this listing is contained in a dirinfo file that can be downloaded from the storage server, if it is not locally cached.
- Selecting a file results in the file being opened using the appropriate application. In order to achieve this, the UI component interacts with the download client to determine whether the file is locally cached or needs to be downloaded. In addition, it uses the file mime type (such as “audio/mpeg”) to determine the application that is registered for handling such files. (Both BREW and J2ME provide application programming interfaces [APIs] that allow this information to be obtained dynamically.) It then invokes that application and passes the relevant file name to it.

We have also implemented several user interface enhancements to make navigating folders and files easier.

1. All the subdirectories are listed before any files are listed. The rationale behind this is to assist the user in navigating through the directory structure more efficiently because in a typical large storage device, the number of files is considerably larger than the number of directories. In addition, the subdirectories and the files are sorted alphabetically according to their names.
2. The scrolling action has a built-in acceleration component. When a user continuously presses the “down” (or the “up”) arrow key, then the speed of scrolling keeps increasing until the user releases the arrow key. This allows users to get to the intended file quickly, even if the number of files in that directory is very large.
3. We have also implemented prefix matching that allows users to type the initial characters of the file names to get to the intended file quickly. When the user first types a character, we use binary search to determine the first file that begins with that particular character. On obtaining such

a file, the file listing is automatically scrolled up to that point and that particular file is selected. As the user types additional characters, we use binary search beginning at that previously selected file to determine the new starting file.

In addition to the features described, the UI component allows the user to browse local directories on the phone and select files for uploading to a specific remote directory.

Client for Downloading and Uploading

The client for downloading and uploading is the component that provides the link between the UI component and the remote storage server. This is similar in nature to the file downloader/uploader component in the Windows CE implementation. Essentially, this waits for a request from the UI component and translates as well as forwards the request to the remote storage server. When any file (including the dirinfo file) needs to be downloaded, this client downloads the file to the local cache, then sends a message back to the UI component indicating the completion of the download. In the next section, we describe the protocol used by this client for interacting with the remote storage server.

Client-Server Interaction

The interactions between the downloader/uploader component and the storage server happen in the form of “commands” that the client sends to the server and the appropriate responses of the server to these commands. Essentially, each packet from the client to the server contains certain commands requesting information, and the server responds with the required information. In our implementation, we have used three different types of commands, as described in the following paragraphs. In general, every command uses the following syntax:

`<command-name> arg1,arg2, ...`

Downloading the Directory Listing From the Server

The download of the directory listing takes place when the user navigates through the directory tree of the remote storage unit. The command has the following syntax:

`DD: <full-path name of the remote directory>`

The download of the directory listing happens in conjunction with the cache synchronization mechanism, which is described in more detail in a later paragraph. Upon receiving this request, the server creates a packet containing the complete file listing of the directory. The packet contains information about each subdirectory and file in that directory. For each subdirectory and file, in addition to name information, we provide other information such as file size, modification date, and time information. As discussed later, this information may be necessary to maintain cache consistency. Cache consistency is necessary if the files on the remote storage server are accessed using multiple devices and one of the devices makes a change to a file or a directory.

In order to maintain cache consistency and keep the file listing synchronized with that in remote storage, we piggyback some extra information in the preceding command. Every directory listing is cached in local storage with the additional information of last modification time for that directory. Whenever the UI component receives a request for the viewing of a particular directory, it first checks whether the dirinfo file for that directory is locally available. If dirinfo is present, then the copy present locally is checked for consistency against the version available with the server. This is achieved by adding the last modification time to the preceding DD command. Thus, the full syntax of the command is as follows:

```
DD: <full-path name of the remote directory>  
    <modification-time>
```

On the other hand, if dirinfo is not present, then the modification-time is set to 0.

The server, on receiving the command, compares the modification time sent in the packet with the actual modification time of the directory. It then creates an appropriate response packet and sends it back. In particular, if the client's last modification time is earlier than the last modification time of that directory on the server, then the server responds with a packet containing the complete directory information. This will expunge all the stale data from the client's cache and replace the data with the latest information. Finally, depending on the information sent by the server, the client loads the directory listing from

the cache or the packet sent by the server. Note that the last modification time on the client can be later than the last modification time on the server because of delayed file uploads. (When a user uploads a file to remote storage, it might not be uploaded immediately for various reasons, e.g., a bad wireless connection.)

Downloading Files

The file download command has the following syntax:

```
DF: <full pathname to the file>
```

The server responds with a series of packets containing the file contents. The packet size here is configurable and can be dynamically changed depending on wireless conditions.

Uploading Files

The mechanism of file upload has the following syntax:

```
UF: <full pathname to the file>
```

The client follows the same procedure as the server for uploading the file contents. In this case, the server waits for the client to upload the file completely.

Performance Enhancements

In this section, we describe several enhancements beyond caching to improve the performance of the remote storage as perceived by the user.

Download Performance

In order to improve the download rates for multimedia, such content can be stored in multiresolution formats that allow streaming of lower resolution media, if necessary. In particular, since the viewable area on a phone is considerably smaller than on the traditional monitors, a lower resolution medium will suffice in most cases. Similarly, depending on the bandwidth of the wireless connection, it may make more sense to download a compact low-resolution version of a file as opposed to one that is bulky and has higher resolution. This allows the content to be delivered faster to the phone and hence results in improved performance.

As mentioned earlier, the wireless connection quality can also be used to determine an appropriate

data transfer rate. As an example, in dual-mode handsets, one can use high data transfer rates when the user is connected to an 802.11 access point, and lower rates for cellular connections. In a similar vein, compression can be used to reduce data size when bandwidth is limited and battery power is not a major constraint. In order to improve data communication performance in a wireless setting, we intend to use an alternative transport protocol to TCP. Our new transport protocol distinguishes between packet loss due to poor link quality and loss due to network congestion. The data transmission rate is only reduced in the latter scenario, not the first, the case of poor link quality.

Finally, to reduce the amount of data transmitted over the wireless link, we plan to support direct data transfers between a data source (e.g., Web site) and the remote storage server. For instance, a user will be able to transfer a (large) file directly from a Web page to the remote server without downloading it to the handset. The benefit here is that if a user wants to download a file for access at a later time, transferring the file over the (high bandwidth) wire-line network to the remote server would be much more efficient than downloading it to the cell phone over a low bandwidth wireless connection. Similar optimizations can be used to improve the efficiency of e-mail transmissions with large attachments—attaching large files to e-mail messages at the server will be a lot more efficient than appending them to e-mail messages at the handset.

Cache Management

Our objective is to make remote storage appear as local storage to the user. In order to achieve this, it is imperative that the access times and the reliability of the remote files match those of any local file. This can be achieved through effective caching and buffering algorithms. We now describe some of the caching strategies that can yield significant benefits.

There are three components to our caching mechanism: (1) reactive cache update, (2) proactive cache update or prefetching, and (3) caching for minimizing access times. We discuss each of these strategies in more detail in the following.

Reactive cache update. Since storage, i.e., flash memory, on the phones is limited, only a few megabytes (MBs) are available for cache utilization. This is analogous to computer systems: here, the flash memory acts as the cache and the remote storage device as the primary hard disk. Thus, whenever we need more space in local storage, we need to select the files that should be stored locally intelligently to attain a high cache hit ratio. Following are a few commonly used schemes along with their pros and cons.

- *Least recently used.* This scheme is useful when the file that has not been used for the longest period is unlikely to be used in the future. This might be true for the files that a person views over a span of time such as documents, songs, and videos.
- *Least frequently used.* This policy would give good results if the probability that the user would view a file again were proportional to the frequency of previous visits. In the file attributes, we store an additional attribute that corresponds to the number of times it has been accessed; this count is incremented every time a file is accessed and is zeroed out whenever the file is not accessed for a sufficiently long period. If the cache is full, then the entry with minimal count is removed and the new entry is inserted.
- *Popular counts.* The user might wish to grade files on the basis of what he thinks should be kept in cache for as long as possible. The user might want to keep certain files such as important personal or company documents, e-mail messages, and some audio/video files always available locally. However, these may be moved into remote storage if space falls short for newly downloaded files.

Proactive cache update or prefetching. Proactive cache update can be used as a measure to enrich the user's experience with remote access. For example, if the user is listening to songs, then prefetching the next likely song can result in improved response times for all the songs and will remove the initial delay that the user could experience for each song. The choice of doing this is left to the user since some users might want to reduce their communication costs and be ready to accept delays instead of paying for unnecessary downloads/uploads.

Caching for minimizing access times. When a file is being evicted from the cache, storing the initial kilobytes (KBs) of data locally can vastly improve the response time of the next access of the file. For example, when moving a frequently viewed video to the remote storage device, we can store the initial few seconds of the video so that next time the video is downloaded again, the user does not have to wait long to start viewing the video. Using this scheme, the user will be able to view the video song as soon as he opens it. This also optimizes the download time when a user decides to close a file after viewing its initial contents. Also, since the initial portion is already present locally, it minimizes the communication costs of downloading the entire file. For document files, only a little more than the first page might be stored locally. Hence, we decide whether to store its initial contents and also how much to store on a per-file basis, depending on the type of the file.

Conclusion

To summarize, our virtual storage tool offers the following advantages to its users.

- *Seamless and transparent access.* The end user does not need to be aware of whether the file he/she is accessing is on the handset. In particular, when the end user wants to access a file, he/she doesn't need to check first whether the file is already on the handset and, depending on the result, try to access remote storage. Basically, with our solution the user does not need to track files manually. Our solution makes access to the remote storage device completely seamless and transparent.
- *Reliability and resilience to handset theft/loss.* The user's data management is completely transparent and automated. In other words, all of the user's data are automatically backed up into remote storage. This makes the user's data resilient to loss or theft of the phone or damage to it. In addition, employing RAID on the remote storage server provides highly reliable storage for the user.
- *High performance.* The end user does not have to worry about manually deleting files on the handset to make space for the additional files that he wants to access. Remote storage access and performance are significantly improved

because of our caching and prefetching/streaming algorithms.

- *Unlimited storage.* Since the amount of storage is no longer dictated by a phone's form factor, the end user can use our tool to have access to potentially unlimited (hundreds of gigabytes of) storage.
- *Ability to access from multiple devices.* Our tool allows user access to the data on the remote storage server from multiple devices including cell phones, PCs, and other media. For instance, a PC can be used to download videos into remote storage, which can be accessed on a cell phone during the day and a home entertainment device (e.g., WebTV) at night.
- *Sharing.* Our tool also allows users to share their data easily with family members and friends, so they can share their pictures, videos, documents, or slides with anyone, anytime, and anywhere.

Thus, our solution is a tool for those who want ubiquitous access to their data. The implementation is robust and agnostic to network downtimes and complete network failures. Intelligent caching is done to adapt to the user's needs and file access patterns. The interesting and challenging task is to be able to calculate the network download speed carefully and then adapt the buffering according to it. Memory and processor constraints have also been kept in mind since most of the phones are not as capable as the desktop computers. Only 5 percent or even less memory is used for storing the data structures of the driver.

*Trademarks

BREW is a registered trademark of Qualcomm Inc. Java and J2ME are trademarks of Sun Microsystems, Inc. Linux is a registered trademark of Linus Torvalds. Symbian is a trademark of Symbian Software Limited. Windows is a registered trademark of Microsoft Corporation.

References

- [1] J. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Trans. Computer Syst.*, 10:1 (1992), 3–25.
- [2] Y.-W. Lee, K.-S. Leung, and M. Satyanarayanan, "Operation-Based Update Propagation in a

Mobile File System," Proc. USENIX Annual Tech. Conf. (Monterey, CA, 1999), pp. 43–56.

- [3] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential Benefits of Delta Encoding and Data Compression for HTTP," Proc. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Commun. (ACM SIGCOMM '97) (Cannes, Fr., 1997), pp. 181–194.
- [4] A. Muthitacharoen, B. Chen, and D. Mazieres, "A Low-Bandwidth Network File System," Proc. 18th ACM Symposium on Operating Systems Principles (ACM SOSP) (Banff, Can., 2001), pp. 174–187.
- [5] W. F. Tichy, "The String-to-String Correction Problem With Block Moves," ACM Trans. Computer Syst., 2:4 (1984), 309–321.

(Manuscript approved April 2007)

SUPRATIM DEB is a member of technical staff at Bell



Labs Research India in Bangalore. He received his B.E. degree in electrical and electronics engineering from the Birla Institute of Technology and Science, Pilani, India; his M.E. degree in telecommunication from the Indian Institute of Science, Bangalore; and his Ph.D. degree in electrical engineering from the University of Illinois at Urbana-Champaign. After his doctoral work, he was a postdoctoral associate with the Massachusetts Institute of Technology, Cambridge, before joining Bell Labs Research India. His research interests include Internet congestion control, resource allocation in wireless and wireline networks, network coding, and distributed systems.

ANKUR JAIN is a member of technical staff at Bell Labs



Research India in Bangalore. He received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi. His areas of interest span wireless and cellular technologies, networking, Internet Protocol television (IPTV), artificial intelligence (AI), and hardware.

ANIRBAN MAJUMDER is a member of technical staff



at Bell Labs Research India in Bangalore. He received his B.E. degree in computer science and engineering from Jadavpur University, Calcutta, and his M.Tech. degree in computer science and engineering from the Indian Institute of Technology, Kanpur. His research interests lie in the areas of databases, particularly in data streaming.

K.V.M. NAIDU is a member of technical staff at Bell



Labs Research India in Bangalore. He received his B.Tech degree in computer science and engineering from the Indian Institute of Technology, Guwahati, and his master's degree in computer science and engineering from the Indian Institute of Science, Bangalore. His research interests span the areas of algorithms, multimedia networking, and software systems for mobile handsets.

JEYASHANKHER RAMAMIRTHAM is a member of



technical staff at Bell Labs Research India in Bangalore. He has a B.Tech. degree from the Indian Institute of Technology, Madras, and a doctor of science from Washington University in St. Louis, Missouri. His primary interests are in the areas of high-speed switching and routing and management of next-generation networks.

RAJEEV RASTOGI is the executive director of Bell Labs



Research India in Bangalore. He received his B.Tech. degree in computer science from the Indian Institute of Technology, Bombay, and his master's and Ph.D. degrees in computer science from the University of Texas at Austin. Dr. Rastogi was named a Bell Labs Fellow in 2003. He is active in the fields of networking and databases and has served as a program committee member for several conferences in the database area. His writings have appeared in a number of ACM and IEEE publications, and other professional conferences and journals. His research interests include database systems, network management, and knowledge discovery. His most recent research has focused on the areas of network topology discovery, monitoring, configuration and provisioning, XML publishing, approximate query answering, and data stream analysis.

ANAND SRINIVASAN is a member of technical staff at



Bell Labs Research India in Bangalore. He received his B.Tech degree in computer science and engineering from the Indian Institute of Technology, Bombay, and his master's and Ph.D. degrees in computer science from the University of North Carolina at Chapel Hill. His research interests span the areas of real time scheduling, operating systems, multimedia networking, and peer-to-peer systems. ♦