Remote Storage for Mobile Devices

Jarkko Tolvanen, Tapio Suihko, Jaakko Lipasti, and N. Asokan

Abstract— **The ability to access remote file storage from mobile devices enables a number of new use cases for storing and sharing data. We describe the design and implementation of a Remote Storage Client framework on Symbian OS, the leading smart phone OS on the market. Our work is inspired and informed by previous work like Coda [1]. We describe why Coda cannot be used directly in our target scenarios and environments. We then describe how we adapted the Coda concepts to suit our needs. The advanced features supported by the framework include disconnected operation with whole-file caching and immediate file access (adapting whole-file caching principle to multimedia-centric smart phones). Using this framework, we have implemented Symbian OS remote filesystems based on WebDAV and FTP.**

Index Terms—**disconnected operation, remote file systems, Symbian OS, WebDAV**

I. Motivation

Carol is on vacation. She takes lots of pictures with her camera phone and fills up almost all the available memory. When she comes back home, her phone detects the presence of a high bandwidth Internet access point and automatically moves her new pictures to her Internet photo album service storage space. So when she sits at her laptop and opens her photo album, she can already see the new pictures. She starts her photo editing software and makes a collage from some of the nicer pictures.

Then Carol goes for an evening out and meets Alice. Carol wants to show off the collage she made earlier. In her phone's media gallery application, she chooses the remote storage tab, locates the collage picture and opens it. It takes a few seconds for the image to download over the cellular network connection. Alice is impressed by the collage. However, Carol insists on adding an audio track to the collage and picks a suitable MP3 from her home server, which she then shortens to fit the collage better. Finally, Carol grants Alice access to her Internet photo album. Alice now sees Carol's Internet photo album as a new tab in her phone's media gallery and is able to view and edit pictures.

These types of usage scenarios that involve **storing**, **retrieving** and **sharing** require support for remote storage access in mobile devices. This paper gives a brief overview of our work in this area.

Manuscript received June 23, 2005.

Jarkko Tolvanen, Jaakko Lipasti, and N. Asokan are with Nokia Research Center, Helsinki, Finland (e-mail: firstname.lastname@nokia.com).

Tapio Suihko is with VTT Information Technology, Espoo, Finland (e-mail: tapio.suihko@vtt.fi).

II. Introduction

Data objects like images and videos are typically stored as files in a file system. The scenario above implies that it must be possible to seamlessly access and modify files from several devices. Applications access files using a standard file system API. Therefore it is logical to integrate remote storage access into the file system. Files could either be transferred directly from device to device or by using a network server.

We have developed a framework for accessing remote filesystems with Symbian OS phones. In remote access, a storage server exports a part of its filesystem, which the phone mounts. The filesystem appears as a new drive on the phone. The storage server can be on a centralized network server or on another phone. The advanced features we have developed include disconnected operation, similar to Coda distributed file system [1], but using more widely deployed protocols (e.g. WebDAV [2]). Disconnected operation means that the client device can continue to use cached files even when the server is out of reach. This implies aggressive caching. We have also developed advanced caching policies that attempt to support disconnected operation and high-latency wireless networks by utilizing whole-file caching whenever possible. However we must take into account that ensuring smooth operation of a multimedia centric smart phone sometimes requires quick access to a portion of data.

In this paper, after setting requirements (Section III), we describe the components of the Remote Storage Client framework (Section IV). We examine some specific issues related to the system in Section V. We then give the first experiences in using the framework in a Symbian OS smart phone (Section VI). Section VII describes related work, and finally, Section VIII outlines directions for future work.

III. Requirements

Our use cases with mobile devices suggest a distributed file system (DFS) that supports high data availability with **disconnected operation**. The client devices in our scenarios may use bearers with different characteristics: cellular data

bearers like General Packet Radio Service (GPRS) have low bandwidth, high latency, and may have volume-based charging, whereas local area bearers like WLAN have opposite characteristics. With a view to easing deployment, we wanted to use **standardized protocols**, and preferably not require server-side changes.

At first glance, Coda would seem like a suitable DFS that supports disconnected operation with aggressive caching. However, we decided not to use Coda for the following reasons:

- It does not use standard protocols, and its deployment is marginal outside the academic community.
- It was mainly designed for high-bandwidth, low-latency bearers.

Although we chose not to use Coda as is, we adopted its basic ideas, like optimistic concurrency control and whole-file caching. As we discuss below, we had to refine or adapt Coda notions to suit the constraints of our target environment. WebDAV has been our primary access protocol. WebDAV is an HTTP extension, and as such has quickly gained acceptance and deployment. It uses reliable TCP/IP communication, so we do not have to take care of e.g. packet retransmissions ourselves. Packets using standard HTTP ports (80 and 443) are usually able to pass through firewalls without a need for extra configuration. This design has also some drawbacks: Perhaps most importantly, we miss server-side callbacks that for example Coda and NFSv4 [4] have demonstrated to be a useful optimization in cache consistency management. However, mobile network (like GPRS) operators do not typically allow incoming packets from Internet anyway, but only allow replies that promptly follow mobile device initiated communication. We also miss the high availability that replicated Coda servers can offer. Moreover, by relying on standard file transfer protocols, we can only refer to files by their path names and cannot assume any path-independent identifiers.

Fetching and caching whole files is a natural consequence of the requirement to support disconnected operation. However, applications sometimes only peek at a part of file contents without any intention to consume the whole file. For example, Symbian OS uses a recognizer architecture to figure out the MIME type of a file by a reading a small data sample from the beginning. Media player applications read metadata embedded into the media files like MP3 to present an informative listing of the contents to the user. When Coda-like whole-file caching is used, attempts to read a few bytes from the beginning of the file may result in a long delay before the whole file is downloaded. Therefore, our system should support **immediate file access** features to enable these cases. Thus we can enjoy some benefits of NFS-like systems, which fetch and cache file blocks.

IV. Remote Storage Client Framework

Symbian OS uses micro-kernel architecture. The file systems in Symbian OS are managed by the Symbian OS File Server, which is a user level process. Our Remote Storage Client (RSC) framework integrates remote file storages into the File Server. The framework uses cache in a local file system, so that disconnected operation can be supported. The framework consists of platform components and applications (see Fig. 1).

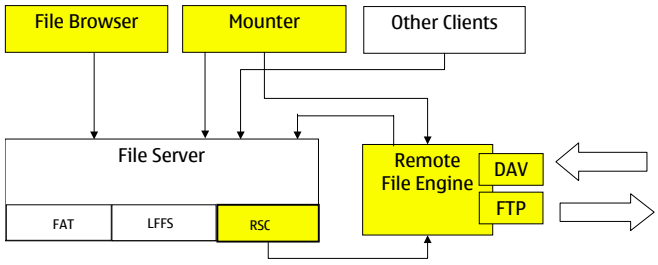

Fig. 1. Remote Storage Client Framework Components

The platform components include a remote file system extension to Symbian OS File Server and a Remote File Engine. Remote file access protocols attach to the engine through protocol plug-ins (WebDAV and FTP in the figure). For mounting and remote file management we have implemented a separate Mounter application and an enhanced Filebrowser.

A. RSC File System

File systems in Symbian OS operate within the File Server. Typically, FAT or LFFS file system is used for local media, like the phone's flash memory or a removable memory card. Each filesystem is mounted on a drive letter. Requests for different drives are handled by different File Server threads. The File Server also offers an interface for dynamically installable file systems to support access to other media, like remote file storages over a network. Remote file system code is also run in a dedicated thread, which makes it possible to recur to the services of the local file systems from the remote file system code. This makes it possible to easily cache remote files.

Our RSC File System attaches to the above-mentioned File Server's interface for dynamically installable file systems. Its design was originally inspired by Coda. In Coda implementations for UNIX variants, the operating system kernel contains a stub file system interface that delegates remote file access operations to a user-space process (called Venus) that performs all communication with remote storage servers. We implemented a corresponding architecture where the RSC module translates operations on Symbian-defined file system objects into file access primitives that are similar to those used at Coda's Kernel-Venus interface. These primitives

define the interface between RSC File System and the Remote File Engine.

In Coda the likely motivation for this separation is that only a small part of the code runs in a privileged kernel mode, and most of the code is easily portable user level code. In Symbian micro-kernel architecture File Server is also a user-level process. However, this separation also has a security benefit in that the Remote File Engine does not need all the privileges of the File Server. This is relevant because Symbian OS version 9 introduced a security architecture where different processes can be assigned different privileges. The File Server is a part of Trusted Computing Base (TCB) and has significant privileges. TCB is kept as small as possible. By separating the Remote File Engine from the File Server, we avoid the need for granting TCB privileges to the Remote File Engine.

B. *Remote File Engine*

The Remote File Engine takes care of remote file fetching and caching. It has been implemented as a Symbian OS server, i.e., as a separate process. This solution adds the process-switching overhead related to client-server communication but, on the other hand, it is flexible by allowing specification of new APIs to the engine's remote access functions that could not be reachable through the Symbian OS File Server API. For example, the engine provides a dedicated API for controlling mounting operations. This API is used by our Mounter application (as indicated in Figure 1).

When remote file storage is to be mounted the engine receives connection parameters in the form of a URI, which, by definition, can convey all information that is needed for attaching to a storage server: file access protocol (service name), server name, TCP/IP port number, username, password, and root directory. For each new mount, the engine instantiates a management object, which then on demand loads a protocol plug-in module that implements the file access protocol specified in the URI.

A protocol plug-in module in our framework is a polymorphic DLL that implements our access interface. The interface consists of primitive file system operations. If a remote storage can be accessed in a meaningful way using this interface, it can be presented as a remote drive in Symbian File Server using this framework. We have implemented protocol plug-in modules for WebDAV [2] and FTP [3], and for accessing the phone's local filesystem (C-drive) through the Remote File Engine. The local filesystem plug-in is used only for testing.

C. *Enhanced Filebrowser*

Typical user interface applications for such system as described here include a mount management application, which allows the user to define mount configurations and mount and unmount remote storages, as well as applications used to access the mounted remote storages. The basic application is a File Browser.

A guiding principle in Coda was *location transparency*: the user does not have to know whether she is accessing a file directly from the server, or from the local cache. We argue that this is not a reasonable assumption when the communication bearer has low-bandwidth or has volume-based charging. Exposing caching via the user interface is a somewhat controversial issue. However, usage of remote files involves first downloading them from the network. In mobile networks, this can be a time consuming and relatively expensive operation. Therefore, it is important to point out to the users that the file in question actually resides on the network.

We have enhanced a file browser for Symbian based S60 smart phones with features that make the user aware of the caching state of files and, to some extent, allow the user to control caching. Our enhanced Filebrowser indicates which files or directories are remote and which have already been fetched and locally cached (the antenna sign in Figure 2).

Fig. 2. Enhanced Filebrowser

Our rudimentary caching control allows the user to explicitly fetch selected files into the local cache. The user can also refresh a cached file or mark a file with a sticky tag that makes the file persistent in the cache.

V. Design Issues

A. *Cache Consistence*

Remote Storage Client should retain cache consistence in cases where multiple client devices access the same files at the same time. When a file is fetched from the remote storage and cached, the local copy in the cache becomes stale if the file in the remote storage changes. Moreover, if also the local copy then changes and is eventually transferred back to the server, a conflict will occur.

Cache invalidation can be avoided if the file in the remote storage is locked. However, locking is overkill when a file is only opened for reading. If locking is not used, the validity of the cache should be guaranteed by some other means. For example, the remote file server could notify the client of changed files. This mechanism is used in Coda, where the

server commits to send the notification, called "callback promise". Also NFSv4 uses a variation of this mechanism. Alternatively, the client could check cache validity when the lifetime of the cached data expires (such mechanism is used in earlier versions of NFS, for example). However, while the client is disconnected from the server and the client is optimistically allowed to access cached files, cache validity cannot be verified and conflicting updates cannot be detected (this is discussed in Section C).

We have implemented a freshness interval based polling mechanism that only involves the client side. Each time a cached copy is opened, and if the validity of the cached copy has expired, the validity is checked against the server. When unique file instance identifiers (for example ETags in WebDAV) are available, those are used to compare the local copy and the file on the server. If the identifiers do not match, a fresh copy is fetched from the server. WebDAV uses the HTTP GET method, which can include the ETag of the cached copy. Only if the client's ETag is not current, the server returns the current file (with its new ETag). In this case a single round trip is required for checking and, if required, getting a fresh copy. Alternatively, the time stamp of the last modification of the file could be used for detecting file changes at the server. This is the only method that can be used with protocols that do not have any explicit support for version comparison, like FTP. The freshness interval defines how long after last checking with the server the cached copy is deemed to be valid without a new check. In mobile networks the freshness interval should be in the upper range of the values used typically in fixed networks, for example 30 seconds for files and 60 seconds for directories (reflecting the lower risk of concurrent updates). Also, this value should be configurable, as network properties such as cost and latency may vary highly.

We have avoided changes to server-side implementations. Server callbacks look like a promising way of increasing the scalability of the system. Protocols for extending WebDAV with such notifications have been proposed [5] [17]. It remains to be seen how this work will progress and how widely support for this feature will be implemented in WebDAV servers. As was mentioned above, configuration of mobile network operator's firewalls may also need to be changed to allow server initiated communication to mobile devices.

B. Immediate File Access

We have relaxed the original whole-file caching principle so that instead of always caching the whole file, we can build the cache file from continuous blocks starting from the beginning of the file and proceeding towards the end. There are good motivations for this in multimedia-centric smart phones. For example, the song title and artist name contained as metadata within an MP3 file should be available even if the song will not be played, so that a music player application can present an informative song list to the user. When the song is then selected for playing, the playing should ideally start before the whole file is fetched and cached.

If the access protocol supports it, we build the cache file in continuous blocks. In principle this means that when receiving a read() request, we first fetch a certain amount of bytes from the beginning of the non-cached portion until the request can be fulfilled. Finally, when the file is wholly cached, it is marked to be available also in disconnected mode. For example, WebDAV uses the Range-header of HTTP GET for this partial reading. We call this feature Immediate File Access, as opposed to pure whole-file caching. An important aspect to consider is the high latency of wireless networks. Because of the high latency, it is not feasible to fetch a remote file at exactly the same granularity that some application uses to read more data from a file. So instead of fulfilling the byte range requested by the application, we attempt to fulfill the purpose of this read with minimal number of round trips.

This can be generalized in the following way: in order to minimize the number of roundtrips needed to read a file, we always fetch the file up to the next threshold. Threshold here is defined as a point beyond which the file "probably" is not needed (yet). In Symbian OS the first threshold value should be 128, because the MIME-type recognizer architecture is used so often, that it doesn't make sense to ever fetch less (the sample value is actually configurable, but 128 is the default value that is used almost exclusively). For example, when the Nokia 6600 Image Gallery application opens a certain JPEG file, its first request is to read 32 bytes from the beginning of the file. In this the case the first read fetches 128 bytes. We avoid a long round-trip to the server if the next request is for bytes 33-127 but lose only a small time in reading the first 32 bytes.

We have defined the next threshold after 128 to be MIME-type dependant. It should cover possible metadata located at the beginning of the file. After reading the first 128 bytes, we have a good probability of knowing the MIME-type of the file, either from the directory (collection) attributes information sent by the WebDAV server (similar to "content-type" header in standard HTTP) or from the Symbian recognizer architecture. For each MIME type we have a threshold value, which tells how many bytes from the beginning of the file we assume to be metadata, and where the actual data begins. This value is read from a configuration file. In addition to the MIME type we take into account device application behavior for that MIME type. For example, MP3 tag (ID3 tag) can in theory be of arbitrary length, but we observed that the Nokia 9500 Media Player reads the first 8000-9000 bytes of the file in order to display the song metadata, so we can set this limit to 10 000 bytes. The configuration file allows us to easily tune the threshold value for each device. It is enough for this value to be a rough estimation. So with MP3 files the second roundtrip would read the bytes 129-10000. This data allows music players to use that MP3 file in a list of songs, and show

metadata such as artist, song name etc., without fetching the whole file that can be several mega bytes of size.

Note that we could of course apply a more deterministic approach, by reading a metadata length header, and then the metadata itself. This would require two roundtrips to get the metadata, but would allow us to read exactly right amount of variable length metadata without considering things like specific device application behavior. The penalty of the extra roundtrip is not worth paying, unless the length of the metadata can greatly vary. In practice we have noticed that this is not the case at least currently.

Finally, when we assume that the read has proceeded beyond the metadata to the actual content data, depending on the file MIME-type we may want to read the data itself in a mode that approaches streaming (again, a typical example is an MP3 file). More specifically, we perhaps would like to return control to the application as soon as possible and fetch the remaining data in the background (read-ahead caching). The initial read can be equal to the number of bytes the application has requested from the File Server, if the network connection has high bandwidth and low latency or the application indicates that it does its own buffering. Otherwise we should read a little bit more (e.g. using formula $f(x) = x + C$, or $f(x) = n*x$, where x is the number of bytes originally requested, C is a constant and n is a factor between 1 and 2).

Sometimes the MIME-type specific metadata is located at the end of the file. For example, the original MP3 tag used the last 128 bytes of the file as metadata. As was mentioned above, we want to keep the cache file continuous. Because of this, a read that goes beyond the currently cached portion will by-pass the cache, i.e. the requested data will be fetched but will not be added to the cache. In practice, we have noticed that the requirement for continuous access is not too limiting and simplifies cache management. With multimedia files sequential access is far more common than random access and in order to support streaming, metadata is usually at the beginning of the file.

C. *Disconnected Operation*

1) *Disconnected Access*

Support for disconnected operation allows the user to access files even when there is no network connection to the remote file server. The following requirements have to be taken into account:

- Connectivity awareness: the system must be able to adapt to varying levels of connectivity.
- Cache Persistence: cached (meta) data must not be lost in system shut down (whether graceful or not)
- Cache Populating (aka "hoarding"): there should be means for deciding on the contents of the cache and mechanisms for populating the cache (e.g., before voluntary disconnection).
- Reintegration: when the remote file server becomes available, the changed objects in the local cache must be synchronized with their replicas in the remote store – this includes conflict detection and resolution.

As a prerequisite for disconnected operation, the system has to be aware of the state of the connection to the remote server. We define three modes of connectivity: strongly connected (like WLAN), weakly connected (a connection with high latency, low bandwidth, or high costs, like GPRS), and disconnected. In weakly connected mode, cache misses are served by fetching data from the remote file server. However, locally updated files are not propagated to the server. This is similar to the "write-disconnected mode" in Coda.

The contents of filesystem objects are stored in the cache as normal files. However, in a system crash, the related metadata would be lost unless it was stored persistently. Therefore, we maintain a persistent copy of the metadata on the disk. The metadata of changed filesystem objects is flushed to disk after each operation on the interface between RSC File System and the Remote File Engine. The size of each externalized metadata entry is typically around 100 bytes.

The client's cache, which has a configurable maximum size, is filled with files as they are fetched from the remote storage. On-demand cache filling is controlled with a prioritized LRU cache management scheme that operates across all mounts. However, before a deliberate disconnection from a storage server, the user may want to download the files that he/she will work with while disconnected. We support this feature, called (explicit) hoarding, by allowing the user to classify selected files as having a higher priority. More sophisticated hoarding could be implemented by using a Coda-like "hoard database" that lists the important files or directories.

Reintegration is performed when a volume in the Remote File Engine moves from a (write-)disconnected state to strongly connected state. The purpose of reintegration is to propagate the client-side modifications to the server. Downloading for local cache update could be included as part of the reintegration. However, we currently rely on on-demand fetching of fresh server contents. Because the server is a standard WebDAV (or FTP) server, the whole procedure has to be conducted by the client. In the reintegration phase, WebDAV locking can be used on the relevant objects to prevent collisions with other clients.

2) *Recording Modifications while Disconnected*

On reconnection to the server, the current states of the client and server replicas could be compared to find out the differences between them. Such a "state-based" comparison would be inefficient if only a small fraction of cached files has been changed, and it would leave certain conflicting updates ambiguous (e.g., we would not know whether a file that only exists in one replica has been created on that replica or deleted from the other one). Our approach is "trace-based" [6], but only at the client side. In comparison, Coda is also trace-based; the client maintains a modification log, but reintegration is

essentially performed at the server that processes the log.

In disconnected mode, Remote File Engine records effects of mutating filesystem operations as part of the objects' metadata. Each object carries an indication of mutations that have been performed on the object and history information about the state of the object that prevailed before disconnection. The history information includes identification of the object's original parent and its path, the object's name, and version information (last modification time, size, and unique file instance identifier if available). This state of an object is saved when it is about to change for the first time. The recorded state is later needed for conflict detection and for generating appropriate actions at reintegration. Our rudimentary operation recording does not preserve the order of the operations, as a fully-fledged modification log would do.

3) Conflict Resolution

We model conflicts between partitioned replicas similarly to Coda [7] and Ficus [8]. However, in our case, the client and server replicas are not symmetric in terms of modification logging, and the node identifiers of files at the server are not visible to the client. Therefore, it is not feasible for the client to find out the original name and location of files that have been renamed and/or moved at the server. Instead, such files appear as new files, and the original files appear to have been removed. Consequently, we have a conflict matrix along the dimensions of file operations on filesystem objects at the client (create, update, remove, and rename) and at the server (create, update, and remove).

The types of directory conflicts are the same as the types of file conflicts, except that name conflicts only appear with Rename operations, and update/update conflicts are assumed to be registered as conflicts in the immediate descendants

Conflict detection is based on client's original and current cache state and the remote file server's repository state after reconnection to the server. To detect conflicts, the client preserves the original state of each object that it modifies. Then it is possible to check whether the server has modified the same object during disconnection.

We require that, whenever possible, resolution should be transparent to the applications and automatic without needing user intervention, even though the user should be notified or otherwise become aware of resolved conflicts. The current implementation produces a human-readable log of the detected conflicts and the results of the automatic resolution.

Conflict resolution applies the "no lost update" principle, where local and server-side updates to files and directories must not be lost. Still, when applying this principle we can have variation in the outcome of the resolution. This variation is controlled by a resolution policy, which can be

- server copy dominates
- client copy dominates
- last writer wins

Because updates must not be lost, there will be situations where we have two instances of the "same" file system object. One of them should be treated as the primary copy while the other one is the secondary copy. The resolution policy only determines whether the server's or the client's copy will be the primary copy.

When "last writer wins" policy is used, the clocks of the parties have to be synchronized. This policy is only applicable if a comparison can be made between both replicas, which is not true for files that have been deleted from the server. Then we must fall back to either of the two dominance policies. In a conflict, the primary copy will retain its name, whereas the secondary copy will adopt the name of the primary copy with a specific suffix and version number appended to the name. For example, the secondary copy of "xyzzy.txt" could be renamed as "xyzzy_conflict_01.txt". The reintegration procedure produces a log of synchronization operations and resolved conflicts. If the automatic resolution has not been satisfactory, the user has to make any desired corrections manually.

VI. Experiences with Using the Framework

We have demonstrated use of remote file systems with various types of settings, like

- Mounting a WebDAV or FTP repository from a network server (e.g., over GPRS or WLAN).
- Mounting a WebDAV repository from another phone (over Bluetooth PAN/WLAN).

The fundamental idea in promoting access of remote resources through the File Server functions has been to allow applications to access those resources with the same existing APIs that they already use with local file systems (access transparency). The location of the resources would then be transparent to the applications, and adapting them to remote file access would only require minor changes, if any.

However, as we have noticed with the enhanced Filebrowser, the network delays cannot always be ignored at the user interface (Section IV.C). Also, the Application architecture or the applications themselves may falsely assume that file access is always fast.

A. Synchronous File Server API calls

An important design decision in Symbian OS has been to optimize the system for event handling. Application framework is such that each native application is a single event-handling thread. Inside the thread so-called active objects are used to handle events non-preemptively.

The main stumbling block in the access transparency ideal has perhaps unsurprisingly turned out to be the fact that many applications exploit the assumption that file operation functions are fast by calling them synchronously. This leads to simpler programming, as the application programmer does not have to implement asynchronous waiting for file system operations. Moreover, the current Symbian File Server API does not even have an asynchronous variant for all the functions.

Alas, calling a long-standing operation synchronously is disastrous for a single-threaded UI application in Symbian OS, as the wait will happen inside a non-preemptively scheduled active object, and will thus block all the active objects in the sole thread. The application cannot handle other events. Important events that it must handle come for example when the application must go to background (user wants to switch applications, an incoming call, etc.).

To alleviate this problem we have modified our framework so that remote file operations which cannot be called via an asynchronous File Server API function take as little time as possible (e.g., call to open() checks that the file is available at the server and the user is allowed to open it in the requested mode, only read(), for which there is an asynchronous variant, fetches the actual data).

The strict requirement for timely response is inherent in Symbian OS. Therefore, the programming guidelines recommend using asynchronous operations when performing tasks with potentially long delays. Consequently, the use of asynchronous system calls is a norm in network socket programming, for example. Many applications already use also asynchronous file operations, as there are situations where even accessing local file systems may take several seconds.

B. Performance

We measured throughput of the Remote Storage Client with a Nokia 9500 phone by using WLAN (802.11b). We accessed files on a Linux laptop (IBM ThinkPad 600) that was running a WebDAV-enabled Apache 2.0 server. We performed reading and copying of a single file with varying size. Each measurement was repeated three times and averages of the measured values were computed. For comparison we made measurements with both a "cold" and a "hot" cache. The results were compared with the phone's native filesystem (C-drive) and with a Remote Storage Client framework loopback mount where the C-drive is mounted through the framework as a "remote" filesystem.

The results of the measurement are shown in Figures 3 and 4.

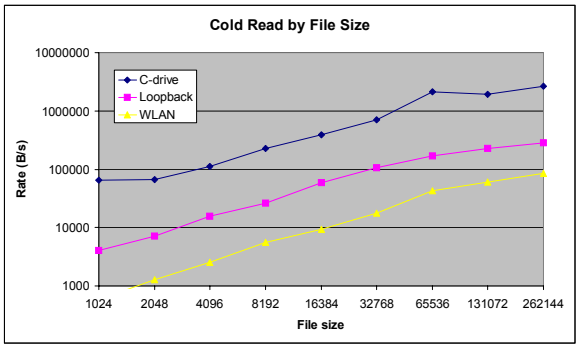

Fig. 3. Cold Read

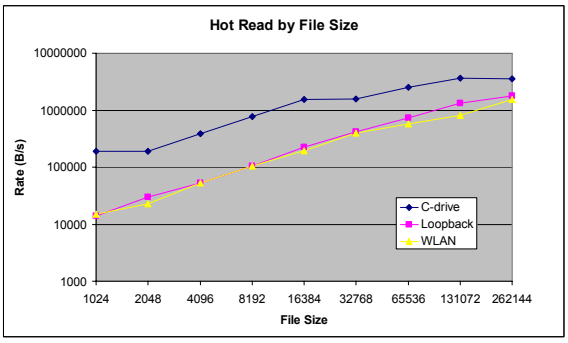

Fig. 4. Hot Read

The cold read throughput of a 16384-byte file was 8-9 kB/s via WLAN. The throughput is better with larger files and poorer with smaller files. With a file size of 1024 bytes the throughput degraded to 155 bytes/second. With larger files the rate approached 100 kB/s. The C-drive curves in the two figures should be identical. The variation is mainly due to the tick granularity (1/64 s) of the system's timer that was used in the measurements.

Hot read uses cached files, so it is independent of the transport medium but shows the caching overhead. Since caching uses the File Server client API, reading a cached file results in two passes through the API.

WebDAV's use of XML makes it a verbose and thus bandwidth consuming protocol. More importantly, in order to optimize throughput in high-latency wireless networks, the number of round-trips should be minimized. In our experimental runs, the round trips typically consisted of queries of the file state by using WebDAV's PROPFIND method in addition to the actual file operations. We still have some room for optimization. For example when the remote storage client framework receives delete() request from the File Server, it first sends via the WebDAV protocol plug-in PROPFIND to check whether the file exists on the remote server and only then sends DELETE to delete it. The lesson learned is that each protocol plug-in implementation must be optimized by carefully examining when the error codes returned by the server for certain operation can be directly mapped to the File Server API error codes, and when some extra queries are needed. For example in this case it is enough to just send WebDAV DELETE operation, as HTTP status codes have enough information to distinguish between the most important File System API errors for this operation. For example "404 NOT FOUND" becomes Symbian system error "KErrNotFound" and "423 FORBIDDEN" becomes "KErrAccessDenied".

WebDAV file locking was not enabled during the measurements, although in practice, a file that is being copied to the server should be locked at the server until the file has been fully copied. Therefore, actually, an empty file is put on the server in order to have some resource to lock before putting the file contents. Some servers may allow creating a

file by setting a lock to a non-existing resource, which would obviate putting the empty file.

VII. Related Work

A standard way to access remote storage is via a DFS. One of the well-known DFSs with disconnected operation support is Coda [1], which is a derivative of Andrew File System (AFS). On each Coda client, a cache manager, called Venus, manages persistently cached whole files and directories. The clients can access cached filesystem objects in a disconnected mode and fetch files in a weakly connected mode. In disconnected mode the updates are immediately applied to the locally cached objects and also logged for later propagation to servers when network connectivity is restored. In the weakly connected mode, the updates can be slowly sent to the server with a mechanism called trickle reintegration. The remote servers that can be replicated for higher performance and fault tolerance, reintegrate conflicting updates originating from different clients.

The Little Work project [9] added disconnected operation support to AFS. The design approach is similar to ours. That is, there was a relatively widely available distributed file system that had to be made usable by mobile clients even when there was no connection to the server – and this had to be done without modifying the servers. Consequently, the DFS client is solely responsible for disconnected operation. Then conflict management, and populating the cache and keeping it persistent become issues for the client. Little Work resolves conflicts automatically by renaming the conflicting entities and putting orphaned entities in a central "orphanage". We also automate conflict resolution by renaming. However, we try to reconstruct removed directory paths for orphans.

InterMezzo [10] is a newer DFS that is inspired by Coda. It utilizes existing journaling file systems (like ext3 in Linux) in transaction handling and guaranteeing consistency. Synchronization is achieved by using a special program called InterSync, which processes a modification log and fetches modified files to the server by using HTTP. Synchronization can be symmetric with both the client and the server device running a web server. By virtue of the layered approach, Intermezzo is relatively simple, and it supports disconnected operation, which makes it suitable for mobile environment. A primary design goal of InterMezzo is reusing standard protocols, but its implementation is tightly coupled to Linux, which makes it challenging to deploy it in other platforms.

WebNFS and later NFSv4 [4] introduced to Sun's popular NFS family of DFSs properties that allow easier use of NFS in Internet. The challenges were similar to those we have targeted; e.g. high latency and access through Internet firewalls. Disconnected operation was omitted from NFSv4 requirements in the design phase [16].

Data synchronization is an integral function in a DFS that supports disconnected operation, but synchronization can also be performed independently of the file system. Such flexible multi-platform utilities include Unison [11] and rsync [12], for example. These user-level programs may optimize data transfers but they cannot take advantage of the modification logging that is built-in in the underlying file systems. Another synchronization framework is SyncML [13], which is specified in OMA (Open Mobile Alliance) for standardizing the way data synchronization is handled on mobile devices. SyncML supports synchronization of any data (expressed as a collection of key-value pairs). The keys are unique identifiers of the objects. If the objects are file-system entities that can only be identified with their path names there should be some mechanism for constructing the unique identifiers and mapping these to path names. SyncML assumes that the client and the server maintain a change log and indicate propagation of logged events with "anchors". The protocol also has provisions for notifying of conflicts during synchronization.

WebDAV has been used as a transport protocol in davfs2 remote file system for Linux [14]. Besides sharing the same transport protocol, we also use a similar implementation architecture, which is inherited from Coda. However, davfs2 does not cache files. Other products that integrate WebDAV into file system in Desktop OSs include Novell NetDrive 6 (part of Novell NetWare and Novell iFolder products) for Windows and Apple iDisk for MacOS. They also support disconnected operation.

VIII. Future Work

Our immediate plans for the future involve areas such as access control, buffering and usability. Reintegration can be improved with more fault-tolerance, background operations (compare with Coda's trickle reintegration), delta transfer of changes only, etc. For access control we are considering implementing WebDAV ACL access control protocol [15] to enable flexible yet safe sharing of information. We also plan to follow the development of proposed protocols that could be used to implement WebDAV server callbacks [5] [17], and possibly adopt one of them, as we see callbacks as a very useful optimization in high-latency wireless networks. A potential usability improvement could be adding multiple mount points under a single drive letter.

The new Symbian OS security architecture brings with it new challenges. Applications allowed to access networking services must have a specific privilege, called the "*NetworkServices* capability", granted to them. Among other things this capability allows controlling which applications may perform operations that cost the user money. Obviously, our Remote File Engine must have the *NetworkServices* capability. Since it acts on behalf of client applications, if it performed no access control on its own, even applications without the *NetworkServices* capability could use remote drives and thus make operations that incur a cost to the user. But, since the client application requests are routed via the File

Server, the Remote File Engine is not explicitly aware of the identity of the client or the set of capabilities the client holds. Thus, the Remote File Engine cannot make access control decisions based on the capability set of the client application. This type of *transitive authorization* is a general problem. To allow a general solution, intermediate servers must pass information about the capabilities held by the clients to downstream servers: for example, the Symbian File Server should pass with each operation the capabilities of the client application to the file system plug-ins (our file system plug-in could then pass them on to the Remote File Engine).

An easy solution then would be to deny remote requests if the client application does not have the *NetworkServices* capability. However, this solution would perhaps be unnecessary restrictive. Note that mounting remote storages happens by calling directly Remote File Engine API (see Fig. 1). In this case the Remote File Engine is aware of the capability set of the client application, and denies the mount request unless the client application has the *NetworkServices* capability. Thus any client application is not able to open network connections to arbitrary hosts using arbitrary protocols, but can only use the existing mounts.

Our framework has two separate privileges, mounting and accessing remote storages, but the privilege granularity in the system does not easily allow making a difference between them. If the main concern is spending user's money, these privileges undeniably become very closely related, and can be expressed as one. Finding an acceptable but not overly restrictive solution to this problem is still an open issue.

IX. Conclusion

In this paper, we have described the design and implementation of a remote storage framework on Symbian OS. We have used this framework to implement filesystems based on WebDAV and FTP. We see WebDAV as a very good protocol choice since, as an HTTP extension, it is widely deployed by existing HTTP servers, yet it provides a good base for building both an online and offline distributed filesystems. Also firewalls usually allow HTTP packets to pass through, so WebDAV can immediately be used in many environments.

Our work is inspired and informed by previous work like Coda [1]. We described why Coda couldn't be used directly in our target scenarios and environments. We then described how we adapted the Coda concepts to suit our needs. The framework is based on a whole-file fetching and whole-file caching scheme. However, it also implements configurable support for "immediate file access". This facilitates operation of applications that get metadata of a file by reading a small part at the head of the file and/or need streaming-type access to contents without first caching the whole file. The system also allows disconnected operation with support for client-controlled reintegration and conflict resolution.

ACKNOWLEDGMENT

The authors wish to thank Seamus Moloney for his help in the early phases of the Remote Storage Client framework development.

REFERENCES

[1] M. Satyanarayanan, "Coda: A highly available file system for a distributed workstation environment", in Proceedings of the Second IEEE Workshop on Workstation Operating Systems, September 1989. Available: http://www-2.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html

[2] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen, "HTTP Extensions for Distributed Authoring - WEBDAV", IETF RFC 2518, February 1999. Available: http://www.ietf.org/rfc/rfc2518.txt?number=2518

[3] J. Postel, J. Reynolds, "File Transfer Protocol", IETF RFC 959, October 1985. Available: http://www.ietf.org/rfc/rfc959.txt?number=959

[4] Brian Pawlowski, Spencer Shepler, Carl Beame, Brent Callaghan, Michael Eisler, David Noveck, David Robinson, Robert Thurlow, "The NFS Version 4 Protocol", 2nd International SANE Conference, May 22 - 25, 2000 MECC, Maastricht, The Netherlands.

[5] J. Hildebrand, P. Saint-Andre, "Transporting WebDAV-Related Event Notifications over the Extensible Messaging and Presence Protocol (XMPP)", IETF Internet Draft draft-hildebrand-webdav-notify-01 (expired), November, 2004.

[6] Trevor Jim, Benjamin C. Pierce, and Jérôme Vouillon, "How to Build a File Synchronizer". Manuscript, February 2002. Available: http://web.mit.edu/6.033/www/papers/unisonimpl.pdf

[7] Puneet Kumar and Mahadev Satyanarayanan, "Log-based directory resolution in the Coda file system", Tech. Rep. CMU-CS-91-164, School of Computer Science, Carnegie Mellon Univ., 1991

[8] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek, "Resolving File Conflicts in the Ficus File System". In USENIX Conference Proceedings, pp. 183-195. Boston, MA, USENIX. June, 1994,

[9] L. B. Huston and P. Honeyman, "Disconnected Operation for AFS". In Proc. First USENIX Symposium on Mobile and Location-Independent Computing, August 1993.

[10] Peter Braam, "Intermezzo: File synchronization with Intersync", version 0.9.3, March 02. Available: http://www.inter-mezzo.org/docs/intersync.pdf.

[11] Benjamin C. Pierce and Jérôme Vouillon., "What's in Unison? A formal specification and reference implementation of a file synchronizer". Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004. Project home: http://www.cis.upenn.edu/~bcpierce/unison/

[12] Andrew Tridgell and Paul Mackerras, "The Rsync algorithm. Tech. Rep. TR-CS-96-05", The Australian National University, June 1996. Project home: http://samba.anu.edu.au/rsync/

[13] OMA SYNCML, http://www.openmobilealliance.org/tech/affiliates/syncml/syncmlindex.html

[14] WEB-DAV Linux File System (davfs2), http://dav.sourceforge.net/.

[15] G. Clemm, J. Reschke, E. Sedlar, J. Whitehead, "Web Distributed Authoring and Versioning (WebDAV) Access Control Protocol", IETF RFC 3744, May 2004. Available: http://www.ietf.org/rfc/rfc3744

[16] S. Shepler, "NFS Version 4 Design Considerations", IETF RFC 2624, June 1999. Available: http://www.ietf.org/rfc/rfc2624.txt?number=2624

[17] Henning Qin Jehøj, Niels Olof Bouvin, Kaj Grønbæk, "AwareDAV: A Generic WebDAV Notification Framework and Implementation", Proceedings of the 14th international conference on World Wide Web, Chiba, Japan, 2005, pp. 180-189.