A Highly Available Replicated File System for Resource-Constrained Windows CE .Net Devices[1]

João Barreto[2] and Paulo Ferreira
INESC-ID/IST
Rua Alves Redol N.º 9
1000-029 Lisboa, Portugal
{joao.barreto, paulo.ferreira}@inesc-id.pt

ABSTRACT

The emergence of more powerful and resourceful mobile devices, as well as new wireless communication technologies, is turning the concept of mobile ad-hoc networking into a viable and promising solution for ubiquitous information sharing. However, the inherent characteristics of mobile ad-hoc networks bring up important challenges for any embedded application developed with the goal of information sharing in the novel usage scenarios enabled by mobile ad-hoc environments. This paper proposes transparent system-level support for Windows CE.Net applications by means of a replicated file system, Haddock-FS. Haddock-FS is based on an adaptable optimistic consistency protocol that provides a highly available access to a weakly consistent view of file, while delivering a strongly consistent view to more demanding applications. In order to effectively cope with the network bandwidth and device memory constraints of these environments, Haddock-FS employs a cross-file, cross-version content similarity exploitation mechanism.

Keywords

Distributed file systems, optimistic replication, mobile ad-hoc networks, Windows CE.Net.

1. INTRODUCTION

The evolution of the computational power and memory capacity of mobile devices, combined with their increasing portability, is creating computers that are more and more suited to support the concept of ubiquitous computation [Wei91]. As a result, users are progressively using mobile devices, such as handheld or palmtop PCs, not only to perform many of the tasks that, in the past, required a desktop PC, but also to support innovative ways of working that are now possible. At the same time, novel wireless communication technologies have provided these portable devices with the ability to easily interact with other devices through wireless network links. Inevitably, effective ubiquitous information access is a highly desirable goal.

Many real life situations already suggest that users could benefit substantially if allowed to cooperatively interact using their mobile devices and without the requirement of a pre-existing infrastructure. A face-to-face work meeting is an example of such a scenario. The meeting participants usually co-exist within a limited space, possibly for a short period of time and may not have access to any pre-existing fixed infrastructure.

If each participant holds a mobile device with wireless capabilities, a mobile ad-hoc network [Cor99] may serve the purposes of the meeting. This way, a report held at one participant's handheld device might be shared with the remaining meeting participants' devices, while its contents are analyzed and discussed. Furthermore, each participant might even update the shared report's contents, thus contributing to the ongoing collaborative activity.

One interesting solution for ubiquitous information sharing is the use of a distributed file system. This approach allows already existing applications to access shared files in a transparent manner, using the same programming interface as for the local file system. However, the nature of the scenarios we are addressing entails significant challenges for an effective DFS solution to be devised. The following lines introduce the main

.NET Technologies'2005 *conference proceedings,*
ISBN 80-86943-01-1

[1] This work was partially funded by Microsoft Research.
[2] Funded by FCT Grant SFRH/BD/13859.

requirements imposed by such challenges that determined the architectural options of our contribution.

High availability. The high topological dynamism of mobile ad-hoc networks entails frequent network partitions. Moreover, the possible absence of a fixed infrastructure means that most situations will require the services within the network to be offered by mobile devices themselves. Such devices are typically severely energy-constrained. As a result, the services they offer are susceptible to frequent suspension periods in order to save battery life of the server's device. From the client's viewpoint, such occurrences are similar to server failures.

These aspects emphasize the need for high availability replication services, so as to minimize the effects of the expectable network partitions and device suspension periods. Pessimistic replication approaches, employed by conventional distributed file systems, such as NFS [Now98] or AFS [Mor86], are too restrictive to fulfill such a requirement.

Adaptability to different correctness criteria. Optimistic replication strategies offer high availability as a trade-off for consistency. While certain applications are able to benefit from such increased availability, some application semantics demand stronger consistency guarantees. In order to be adaptable to a wider set of applications, replicated systems should offer multiple consistency levels: from a relaxed consistency, highly-available to a sequentially consistent mode of replica access.

Adaptation to resource-constrained devices. Whichever strategy is taken, the memory and bandwidth limitations of mobile devices and wireless links, respectively, must be taken into account. For optimistic strategies, the update log is the main memory overhead, while network usage is typically dominated by replica synchronization.

This paper describes Haddock-FS, a transparent replicated file system for Windows CE.Net collaborative applications, including .Net Compact Framework applications. Haddock-FS is based on a highly available optimistic consistency protocol. In order to cope with the resource constraints of mobile devices, Haddock-FS employs content similarity exploitation mechanisms. The paper focuses on the main implementation issues regarding Haddock-FS and the Windows CE.Net development environment. Furthermore, a thorough experimental evaluation using actual embedded devices is presented.

The rest of the paper is organized as follows: Section 2 introduces the main architectural aspects. Section 3 addresses application programming interface aspects, while Section 4 describes the implementation of Haddock-FS. Section 5 presents experimental results. Finally, Section 6 describes related work and Section 7 draws some conclusions.

2. ARCHITECTURE

This section briefly introduces the architecture of Haddock-FS, as originally proposed in [Bar04a].

File System Consistency

Haddock-FS is a transparent, peer-to-peer replicated file system designed to support a broad set of usage scenarios that are made possible by mobile networks. It relies on a hybrid consistency architecture, based on epidemic propagation of replica updates, that accommodates for applications with differing consistency demands: a tentative view, supporting any-time, anywhere read and write access to shared files, at the cost of weak consistency guarantees; and stable view, offering sequentially consistent [Lam79] access to shared files as a trade-off for reduced write availability.

Each Haddock-FS mobile peer constitutes a replica manager that is able to receive file system requests and perform them upon its local replicas. The underlying replication mechanisms are transparent to applications, which may access Haddock-FS's services by using the same API as the one exported by the local file system. Provided the accessed files are locally replicated at a given Haddock-FS peer, the file system services will be available, independently of the current network connectivity.

Update propagation is achieved by pair-wise reconciliation sessions between mutually accessible replica managers, where replica updates are epidemically propagated. Complementarily, Haddock-FS uses a primary commit scheme [Ter95], in which a single replica of each file, the primary replica, is responsible for selecting new stable updates and propagating such decision back to the remaining replicas. Each file is initially assigned a unique primary replica, at which it was originally created. After creation, primary replica rights may be transferred to other replicas, by exchanging a token that identifies the current primary replica.

For each file replica, a replica manager maintains: a stable value, which holds a version of the file's stable contents and an update log, which records the data specifications of most recent update requests that have been accepted by the file replica.

Content storage and propagation

The inherent memory and bandwidth constraints of mobile devices and wireless links are severe limitations to the effectiveness of a distributed file system for ad-hoc environments. For this reason, Haddock-FS tries to reduce the size of update logs stored at each device, as well as of update data to be transferred during replica reconciliation.

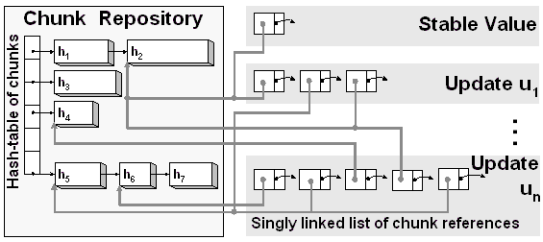

Figure 1. Example of replica content storage.

This is achieved by exploiting the cross-file and cross-version similarities that exist within the replicated data held by Haddock-FS's mobile peers. Such approach is based on that of the Low-Bandwidth File System (LBFS) [Mut01].

The basic idea of the content storage and transference scheme consists of applying the SHA-1 [NIS95] hash function to portions of each replica's contents; each portion is called a chunk. The obtained hash values can be used to univocally identify their corresponding chunk contents. From this assumption, if two chunks produce the same output upon application of the SHA-1 hash function, then they are considered to have the same contents.

A content-based approach is employed to divide replica contents into a set of variable-size non-overlapping chunks, in order to minimize the effect of *insert-and-shift* operations in the global chunk structure of a replica [Mut01].

Haddock-FS extends the use of LBFS's strategy to both local storage and network transference of replicated file data. Our solution considers the existence, on each file system peer, of a common chunk repository which stores all data chunks, indexed by their hash value, that comprise the contents of the files that are locally replicated at that peer. The data structures associated with the content of locally replicated files simply store references to chunks in the chunk repository. This applies both to the update log and the stable value of each replicated file. Hence, the contents of an update or replica value consist of a singly linked list of references to data chunks, stored in the chunk repository (see Figure 1). So, if different files or versions of the same file contain data chunks with similar contents, then they will share references to the same entry in the chunk repository, thus reducing memory usage.

Read accesses to a file's contents can be served by a single indirect memory access to the chunks referenced by the chunk references stored in the file's data structures. Serving a write request upon a local replica is, in turn, a more expensive operation. In order to optimize situations where already stored contents are modified in a partial region, an *incremental chunk update algorithm* [Bar04b] is used. Such algorithm ensures that only a minimum set of affected chunks, from the original contents chunk list, is actually re-evaluated.

Update propagation between peers also makes use of the chunk repositories of each peer. When a chunk has to be sent across the network to another peer, only its hash value is firstly sent. The receiving peer then looks up its chunk repository to see if that chunk is already stored locally. If so, it avoids the transference of that chunk's content and simply stores a reference to the already existing chunk. Otherwise, the chunk contents are sent and a new chunk is added to the repository.

3. Application Programming Interface

Haddock-FS exports the same application programming interface (API) as the standard file system API of Windows CE.Net. Examples of such interfaces are the standard *CreateFile*, *CloseFile*, *ReadFile* and *WriteFile*. Therefore, any existing Windows CE.Net application that is originally built to access the local file system may transparently use Haddock-FS's replicated file services.

In particular, if one considers application programming using the .NET Compact Framework, programmers may continue to use conventional class libraries such as *System.IO.FileStream* or *System.IO.File* to access and manipulate file system objects of Haddock-FS. Since the implementation of these classes relies on the standard file system API, file system objects located within Haddock-FS's name space may be transparently accessed.

Nevertheless, some specific aspects of Haddock-FS's behavior are not controllable by the conventional file system API; namely, the aspects related to the replication protocol. This implies that some extended control must be provided beyond the conventional file system API. Such control should allow users to perform replication operations while running replication-blind applications that solely rely on the conventional file system API to manipulate Haddock-FS's objects. Examples of such operations are switching from a tentative to a stable view of an opened file, and vice-versa, and to transfer primary replica rights to another accessible replica. Replication control should also be granted to programmers that wish to develop replication-aware applications for use with Haddock-FS.

Replication control is provided by means of reserved control codes passed to the standard *DeviceIoControl* interface, also exported by Haddock-FS. The actual calls to DeviceIoControl are performed by a replication control class library, which replaces the interaction with *DeviceIoControl* with a more programmer-friendly interface. Currently, a replication control class library is available for use by .Net Compact Framework applications, which extends the standard

System.IO.FileStream class. Illustrative methods of the class are shown in Table 1.

```
bool switchToTentativeView();
bool switchToStableView();
bool grantPrimaryRights(RepId destRep);
```

Table 1. Example of replica control class methods.

4. Implementation

Haddock-FS is an Installable File System Driver [Mur98] for the MS Windows CE.Net embedded operating system. The current version supports the replica consistency protocol, as well as cross-file and cross-version similarity storage and network usage optimizations. All relevant file system functions are implemented. Interaction between peers is achieved using a remote procedure call library that was developed along with Haddock-FS.

Installable File System Driver

Haddock-FS's API is exported by an installable file system driver (IFSD), in the form of a dynamic link library. Such programming interface is comprised of file system functions, which form the client side of each Haddock-FS's peer. Using the *LoadFSD* function of the FSD Manager service of Windows CE.Net, the file system can be mounted at run time.

The server side of each peer resides in a thread of the Device Manager process that is created when the file system is mounted. The server thread is continually waiting for remote procedure call requests from other peers across the network. Such requests are served upon access to the file system data structures stored in the address space of Device Manager process. On the other hand, the file system functions that are exported by the dynamic link library constitute the client side of each peer. Most of such functions access the shared data structures of the server thread.

4.1.1 Data structures

Haddock-FS maintains a collection of data structures in the address space of the Device Manager process, where the IFSD is loaded. Most of the exported file system functions access and modify such data structures when called. The most relevant data structures are as follows.

- Chunk repository, as described in Section 2.
- Root directory, which contains a hierarchical representation of the file system objects (directories and files) that are currently known by the local peer, including their relevant file system attributes; their creation, modification and access times and read-only, hidden or archive flags. In the case of locally replicated file objects, replication information is also included.
- Open file table, holds entries for the files that are currently opened by some process. Each entry contains information about the current file pointer position, as well as the share mode and access type, specified when the file was opened.

4.1.2 Exported File System Interfaces

The file system interfaces that are exported and implemented by Haddock-FS's IFSD may be grouped into the following categories [Mur98]:

1. Device event interfaces, which handle the initialization and termination procedures of the file system driver. These events correspond, respectively, to the *MountDisk* and *UnmountDisk* functions. Such functions are not available to applications through the file system API. Instead, they are only called by FSD Manager in order to mount or unmount the IFSD. The *MountDisk* function is responsible for: registering a volume where Haddock-FS's shared file system structure will be accessible to applications; initializing the local file system data structures and RPC services; and creating a server thread, which will handle all remote requests from other Haddock-FS peers. Inversely, the *UnmountDisk* function handles deregistration of the file system volume and termination of the server thread.

2. Path-based interfaces, which access or modify file system objects that are identified by their alphanumeric path names when the interface is called by applications, such as *CreateDirectoryW*. Every path-based function first decomposes and analyzes each path name argument so as to locate the corresponding element in the root directory structure. The requested operation is then performed.

3. Handle-based interfaces, which access or modify files that are identified by a previously obtained file handle, such as *ReadFile* or *WriteFile*. A file handle is obtained by a call to the CreateFileW function, in which a path name is passed as an argument to identify the desired file. Additionally, other relevant arguments specify the intended share mode and type of access. Similarly to any path-based function, the supplied path name is used to obtain a reference to the corresponding file element in the root directory structure. If found, the open file table is examined to verify that no sharing conflicts will occur with the current entries in that table. Finally, if such requirement is fulfilled, a new entry is then inserted into an empty slot of the open file table and its position within the table is returned. Such integer value is a file handle that must be used by succeeding calls to handle-based file system functions to the same opened file.

4. Find interfaces, which allow applications to iterate through the list of file system objects whose path name matches a given search string. Namely, *FindFirstFileW*, *FindNextFileW* and *FindClose*.

Remote Procedure Call Library

The developed RPC library is based on the Winsock 2.0 network programming interface and incorporates an interface description language (IDL) and its respective compiler. The IDL allows programmers to specify the remote procedures that will constitute their distributed application (in this case, the Haddock-FS driver itself). Accordingly, the compiler automatically generates program code that allows the distributed application to call and serve the specified remote procedures.

It should be emphasized that no native RPC services are available in Windows CE.Net. Although the available DCOM services of Windows CE.Net are based on an underlying RPC library, its interfaces are not directly available to programmers. Furthermore, the RPC components that support DCOM are reduced to the subset of features that are strictly required by DCOM.

5. Evaluation

Haddock-FS was evaluated through several experiments. All measurements were obtained while running one or more Haddock-FS peers on the Arcom VIPER development board, which includes a 400MHz Intel Xscale-based PXA255 processor with 64MBytes of RAM and a 32MB of an Intel StrataFlash drive. It is worthy to note that such experimental platform provides testing conditions very similar to the memory and processor characteristics found in typical real world settings.

To evaluate Haddock-FS's performance with practical workloads, we used an unmodified version of the MS WordPad word processing application to access replicated files. This application is typically bundled with Windows CE.Net devices.

Chunk Repository Efficiency

The first experiment measured the effectiveness of local replica content storage, based on the use of a chunk repository. We simulated the composition of an actual scientific paper [Bar04a] using 19 different backup versions of its source text, ordered chronologically. The set of backup versions represents the real evolution of the paper, sampled periodically for approximately six months, from an initial version with a few paragraphs to a final version with eleven pages occupying 33 Kbytes. The size of the versions, as well as the character of document is considered to be extremely representative of the documents that are normally accessed by mobile devices. Each version contents were individually applied to a local file replica by using the WordPad application to open, write and close such contents to the replica. The measured optimal expected chunk size for the used workload is 256 bytes, which achieved a substantial reduction of 47% in memory usage by use of the chunk repository, in comparison to a non-optimized approach (that is, without cross-file, cross-version content similarity exploitation).

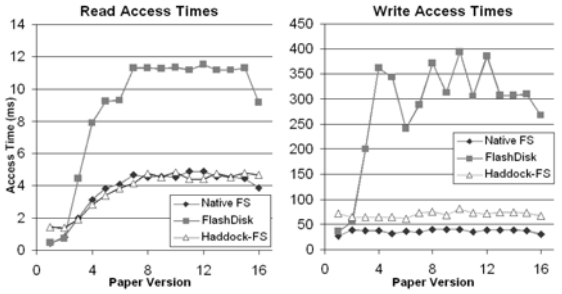

Figure 2. Local access times.

Finally, a more complete experiment was conducted, in which two Haddock-FS peers collaboratively issued updates to a shared replicated file. The considered set of updates was the same as the previous experiment, though distributed by both peers. The obtained results showed that, from a total amount of 460Kbytes that needed to be transmitted during reconciliation sessions between peers upon acquisition of the write token, only 237Kbytes (58%) were effectively sent.

Local Access Times

One experiment measured the impact of Haddock-FS replica storage architecture in the performance of local file system calls. The performance of Windows CE.Net native file system was used as the primary evaluation reference. Furthermore, the performance of a *Transaction-Safe File Allocation Table* file system mounted on an onboard flash drive was also measured.

The experiment was conducted by running a test application that performed and measured the latency of write and read file system calls to different versions of the paper. In order to deal with occasional deviations induced by external factors such as the processor workload, the access time measurements were repeated several times in the same experimental conditions and the average value was then considered.

Haddock-FS read accesses are, on average, 16,5% slower than the native file system, as shown in Figure 2. However, if one considers only read accesses to versions with more than 10KBytes, Haddock-FS actually outperforms the latter by 1,7%.

The measured write performance of Haddock-FS reflects the extended complexity that is imposed by its content similarity exploitation architecture, as shown in Figure 2. Write accesses are, on average, 92% slower than the native file system counterpart. Still, the measured write performance of Haddock-FS is, on average, 75% better than that of the FlashDisk file system. Since most of today's commercial devices are equipped with secondary storage devices

with similar access performance, this evidence suggests that typical mobile users will tolerate Haddock-FS's write access performance.

6. Related Work

The issue of optimistic data replication for loosely coupled environments has been addressed by a number of projects, most of which do not assume that replicas will be held by resource-constrained devices. Bayou [Ter95] is an optimistic database replication system that relies on application-specific conflict detection and resolution procedures to attain adaptable consistency guarantees. The non-transparent character of Bayou's approach prohibits the use of already existing applications, in contrast to Haddock-FS's solution.

The Roam [Rat99] optimistically replicated file system does not require replica managers to store an update log, which eliminates the significant memory overhead that is typically imposed by such a data structure. Nevertheless, Roam's consistency protocol does not regard any notion of a stable replica value. This limitation restricts Roam's applicability to applications with sufficiently relaxed correctness criteria that tolerate dealing only with tentative data.

AdHocFS [Bou03] exploits the high connectivity of ad-hoc groups of replica managers by enforcing a pessimistic strategy amongst the group members. Nevertheless, AdHocFS's architecture is still based on the existence of fixed server infrastructures, where the stable values of files are held. Therefore, should that infrastructure be unavailable, users and applications are restricted to accessing merely tentative data.

Finally, content similarity has already been exploited for storage purposes by the Pastiche backup system [Cox02], so as to minimize storage overhead on backed-up contents. However, Pastiche's file system does not employ incremental writes to chunked contents; instead, each write operation causes the resulting contents to be re-processed by the chunk division process. Though acceptable for back-up operations, such solution may not be adequate for partial content modifications.

7. Conclusions

Haddock-FS is a replicated file system designed to meet the requirements imposed by mobile ad-hoc scenarios, in order to provide a viable support for collaborative activities. Namely, high availability, adaptability to different correctness criteria and adaptation to resource-constrained devices.

Haddock-FS has been successfully implemented in Windows CE.Net and tested in Arcom VIPER XScale-based development boards. Experimental results show that Haddock-FS accomplishes significant network and memory usage reductions when compared to traditional solutions, while attaining acceptable access times.

8. References

[Bar04a] Barreto, J. and Ferreira, P.. A Replicated File System for Resource Constrained Mobile Devices. Proceedings of IADIS Applied Computing, 2004.

[Bar04b] Barreto, J. Haddock-FS: A Distributed File System for Mobile Ad-hoc Networks. M.Sc Thesis, Instituto Superior Técnico, 2004.

[Bou03] Boulkenafed, M. and Issarny, V. Adhocfs: Sharing files in wlans. Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications, Cambridge, MA, USA, 2003.

[Cor99] Corson, S. and Macker, J. Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations. Internet RFC 2501, IETF, 1999.

[Cox02] Cox, L., and Noble, B. Pastiche: Making backup cheap and easy. Proceedings of 5th OSDI, 2002.

[Lam79] Lamport, L.. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, 1979.

[Mor86] Morris, J. et al. Andrew: a distributed personal computing environment. Communications of the ACM, 29(3):184–201, 1986.

[Mur98] Murray, J. Inside Microsoft Windows CE. Microsoft Press, 1998.

[Mut01] Muthitacharoen, A., Chen, B. and Mazieres, D. A low-bandwidth network file system. SOSP, 2001.

[NIS95] National Institute of Standards and Technology. FIPS PUB 180-1: Secure Hash Standard. National Institute for Standards and Technology, USA, 1995.

[Now89] Nowicki, B. Nfs: Network file system protocol specification. Internet RFC 1094, IETF, 1989.

[Rat99] Ratner, D., Reiher, P. and Popek, G. Roam: A scalable replication system for mobile computing. Mobility in Databases and Distributed Systems,1999.

[Ter95] Terry, D. et al. Managing update conflicts in bayou, a weakly connected replicated storage system. Proceedings of the 5th ACM SOSP, 1995.

[Wei91] Weiser, M.. The computer for the twenty-first century. Scientific American, 265:94–1, 1991.