Opportunistic File-associations for Mobile Operating Systems

Umar Saif
LUMS Computer Science Department[1]
umar@lums.edu.pk

Abstract

This paper presents the design and implementation of Opportunistic file-associations, designed to decouple the storage and handling of user files in a mobile device. A file with an Opportunistic file-association is not limited to a local application environment on a mobile device. Rather, the runtime environment supporting Opportunistic file associations causes a user file to be opened at the most appropriate device in the environment of the user.

1 Introduction

In conventional operating systems, such as MS Windows, a file may be associated with an application, typically called its file association. The file association specifies the application that is launched by the operating system when a user opens the file for reading, writing or executing the contents of the file. For instance, a multimedia-file may be associated with a media player, a text file may be associated with a text editor and a compressed archive may be associated with an archiving application.

Traditionally, this model has served well in simplifying the user interaction with the operating system – a user simply "double-clicks" on the desired file and an appropriate application environment is automatically provided by the operating system. However, in traditional operating systems, while there is no restriction on whether the file is stored locally or fetched from a remote file server, the application launched by the operating system is local to the device on which the user attempts to access the file. This is often overly limiting for mobile devices that are typically resource-constrained and offer only limited I/O facilities. Therefore, it is neither always possible nor desirable to launch a local application for reading, writing or executing a user file. For instance, a typical handheld device such as a Compaq iPAQ may be

augmented with a few Gigabytes of Flash memory and a wireless network interface, but it has limited computing resources and battery life, lacks a proper keyboard and offers only a small display and low-power speakers. Thus, such a mobile device may hold a user's pictures, documents and music, it is not ideal for viewing a PDF file, editing a Powerpoint presentation or playing a high-quality video-stream. For instance, most applications in the Microsoft Office suite for Pocket PC (Pocket Office) lack a number of editing features, such as styles, tables, annotations, footnotes, headers and footers, necessitating the use of the full-blown desktop versions to generate nontrivial documents.

Our experience with Oxygen computing environments [1] has shown us that it is often possible to improve the quality of such applications by utilizing the computing resources in the environment of the user, such as a desktop or a laptop computer. In this paper, we propose Opportunistic File-Associations (OFAs) as a mechanism for decoupling the storage and handling of user files in a mobile device. A file with an Opportunistic-file-association is not limited to a local application environment on a mobile device. Rather, the runtime environment supporting Opportunistic file associations causes a file to be "opened" at the most appropriate device in the environment of the user. Any subsequent changes in the environment of the user may revise the file-association, cause the handling of the file to be moved back to the mobile device, or suspend the operation on the file until a time a suitable resource becomes available in the environment of the user.

To put our work in context, the OFA runtime enables a computing model that is exactly opposite to the one afforded by systems such as the Coda File System [2]; while previous systems like Coda enable

[1] *This work was done while the author was at the MIT Computer Science and AI Laboratory (CSAIL)*

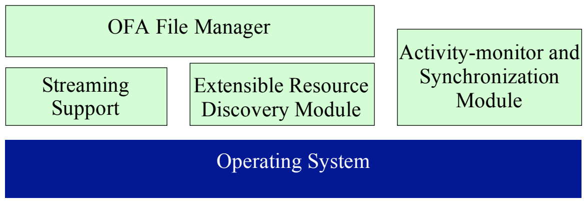
OFA File Manager
Streaming Support
Extensible Resource Discovery Module
Activity-monitor and Synchronization Module
Operating System

Fig. 1. OFA Runtime Environment

files stored on a remote server to be accessed and modified at a mobile client, OFA runtime environment enables files stored at a mobile client to be opened, modified and executed on a remote server, albeit opportunistically. We believe that such a computing model will become increasingly important with the widespread use of "pluggable" memory, such as Compact Flash (CF), MMC and SDM, and wireless-equipped mobile devices, such as Compaq iPAQ. With our system, users will be able to keep all their important data "handy", stored in a memory card plugged into their PDA, while automatically taking advantage of nearby hosts to view and modify that data. Similarly, Opportunistic File-associations complement the popular "synching" applications, which synchronize the data stored on a mobile device with the primary copy on the user's desktop. The OFA runtime environment, on the other hand, opportunistically copies data from a user's mobile device to a nearby host and synchronizes updates to the data at the remote host with the primary copy on the mobile device.

Our current work builds on our experience with Service-oriented Network Sockets (SoNS) [3] and Lightweight Adaptive Network Sockets (LANS) [4], designed to opportunistically connect a stream-oriented application to resources in its environment. Opportunistic File-associations define a complementary architecture for mutable files and do not require any changes to the application code; by provisioning opportunistic resource utilization at the file-association level, rather than the network-socket level, the OFA runtime environment automatically affords compatibility with legacy applications such as MS Word.

2 System Overview

Figure 1 show the architecture of the OFA runtime environment. The runtime environment comprises of four modules: a) OFA File-manager, b) Resource discovery module, c) User-activity-monitor and synchronization module and d) Streaming server and client. Below we describe each of these components in more detail.

The OFA runtime environment is designed as a user-space process and does not require any changes to the underlying operating system. On the client-side, OFA-runtime is designed as a simple, portable file-manager that exports an interface similar to Windows File Explorer. Using the OFA file-manager, a user can browse the files stored on a mobile client and choose a file to be opened by the OFA runtime environment.
The OFA runtime environment extracts the extension (type) of a chosen file to discover a suitable host for opening the file. Matching hosts are discovered by the OFA extensible discovery module, modeled after the framework in SoNS [3]. Matching hosts are prioritized based on the device's computing and I/O capability e.g. display size, keyboard, mouse, processor speed, current load and available memory. These attributes and their importance in determining a suitable match is configurable by the user (as part of a special configuration file ofa_config.conf) and may be customized for each file extension.

Once invoked, the discovery module also starts periodically probing the environment for changes in available hosts and invokes the runtime system if there is a change in the user environment. If a change in user environment is detected by the system, it updates the file-manager's list of candidate hosts for each open file. This list may also be viewed by the user at any time to manually select an alternative host/application, causing the system to migrate the session to the new host. A user may also configure the system to automatically migrate a session if a newly discovered host matches a pre-specified criteria (explained later).

The OFA runtime environment employs rsync, a diff-based efficient file-synchronization framework, for replicating and synchronizing user files on remote devices. Large multimedia files, however, are streamed to the remote computer by starting a streaming server on the mobile device in response to the file-open event rather than incurring the cost of copying a large file over the network.

Fault-tolerance in our system is achieved by periodically monitoring the health of the remote host used for opening a user-file. If a remote host fails to respond to a health probe, the runtime environment migrates the session to another device in the environment of the user. If an appropriate device cannot be found in the environment of the user, the runtime environment fails-over to the mobile device. For file-types that cannot be opened on the mobile device, and hence do not have a local file-association, the OFA runtime environment simply saves the current state of the file and suspends the session for a later time. The OFA runtime environment employs SSL to protect against eavesdropping of user data over the network.

Below we briefly discuss the key architectural considerations for the OFA runtime environment.

2.1 Layer of Abstraction

As part of Project Oxygen, we have experimented with two different approaches for accessing resources in the environment of a mobile device: Intentional Naming System (INS) [5], Service-oriented Network Sockets SoNS (and LANS) [3][4]. Both these approaches integrate resource discovery and remote access with the underlying communication primitives; INS's late-binding architecture combines dynamic resource access with the network routing layer, while SoNS (and LANS) provides opportunistic service access at the network-socket layer.

However, our experience with SoNS and LANS has shown us that a wide-range of simpler applications, such as "open this document for editing on the closest desktop, if one is available", have three unique requirements not suitably addressed by an "intelligent" messaging layer. First, such scenarios typically involve shrink-wrapped, legacy applications, such as MS Word, that cannot be modified to use a new messaging API. Second, beyond initial discovery and selection of a suitable resource in the environment of the user, such applications typically do not require an aggressive re-binding of the application-sessions. Finally, such a system must ensure consistency of data if the user edits the contents of the file on a remote device. The approach presented in this paper addresses this class of applications.

By interposing a redirection layer at the file-association level, the OFA runtime-environment "works-around" an existing application; instead of requiring an application like MS-Word to be re-written using an "intelligent" messaging layer, the OFA runtime layer simply moves the user data to an appropriate remote device, uses an existing application to open the file and provides a mechanism for maintaining consistency between the mobile device user data.

2.2 Automation vs. User-control

Perhaps the most controversial, and arguably the most interesting, aspect of our research is "automation"; the goal of our research is to reduce the tedium of manual discovery, configuration and maintenance of services in a pervasive computing environment. Opportunistic file-associations, as well as our previous two systems, SoNS [3] and LANS [4], are designed to *automate* the process of spontaneous discovery and utilization of devices that become available in the environment of a peripatetic user.

However, such automation is inherently in tension with user-control; though there is some clear advantage in automating the task of discovering available hosts and making "reasonable" choices for offloading user-files, it is also important for users, on the other hand, to have sufficient control over the system to avoid "unpleasant surprises".

We believe that the practicality of such a system depends on striking a balance between automation and user-control. The rule-of-thumb we followed in our design was to err on the side of caution when automating; by default the system simply aids in the manual selection of remote hosts for opening files. However, the system also exposes a number of control parameters that may be configured to specify the degree of automation by the system.

In its default behavior, the OFA runtime environment periodically discovers available hosts (and matching applications) and maintains a list of matching hosts/applications for each file-type. A user can view this list via the OFA file-manager and manually select the host/application for opening a file. Selection of a (new) host/application for an already open file causes the file to be moved to the new host.

In some circumstances, this process may be automated without adversely affecting the usability of the system. For instance, if a user repeatedly offloads PDF files on his PDA to his laptop for viewing, the system may automatically perform this operation next time a user wishes to open a PDF file on his mobile device.

Our system implements such automation using location-based "historical hints". When a user first wishes to open a file, the system simply prompts the user with the option of opening the file either locally or remotely and presents her with a list of matching hosts/application. In turn, the system stores the user selection as the default file-association for that location. When the user next wishes to open the file at the same location, the system automatically chooses the previously selected host as the default option. If a previously selected remote host is not available, the system, based on the knowledge that the user preferred to open the file remotely, automatically offloads the file to the best available similar host/application, if one is available. (If a user is dissatisfied with the default choice, she can bring up the OFA selection-menu and select another resource, causing the file to be moved to that resource while updating the default file-association for next invocation). By indexing default file-associations by location (by for instance using an indoor location system), our architecture permits multiple defaults to be set, for instance one for the user's home and one for her office. This simple scheme has obvious appeal. For example, the OFA runtime environment automates the task of a user who keeps her pictures on her iPAQ CF-card but prefers to view them on her office desktop when she is in her office. At the same time, such a scheme avoids a scenario where a user's music, stored on her portable MP3 player, is automatically streamed to her laptop even when she prefers to listen to the music on the headphones connected to her MP3 player.

The practicality of our approach also depends critically on contextualizing the process of automatic host selection and utilization with respect to the user; a user must be able to specify the context within which the system establishes file-associations to offload files. Such specification of user context is important to avoid scenarios in which, for instance, the system offloads a user file to the desktop in his office when he opens the file in his car.

The OFA runtime environment permits a user to specify a context for each file-type. Context may be specified as a network address or location of the looked-up host. For example, the current implementation uses the SCOPE parameter of an SLP (IETF Service Location Protocol) network query to set the context of discovery. It is also possible to use a location system such as the MIT Cricket Location System to set the location of discovered resources, or use the Bluetooth Service Discovery Protocol (SDP) to constrain host discovery to a small radius around the mobile device. We are currently exploring mechanisms for more fine-grained and precise specification of context, such as the use of pointing devices like the WorldCursor [13].

An alternative to our approach could be to export the file-system on the mobile device such that a user can browse it from a remote host and manually select the file to be opened at the remote host. While this approach circumvents the problem of contextualization, it requires manual intervention from a user. Such manual intervention is justifiable only in some circumstances, for instance when a user intends to edit the selected file after opening it. In most other scenarios, such as streaming applications like follow-me-video, *controlled automation* is still preferable.

In addition to *context*, the OFA runtime exposes three more parameters that can be configured to control the behavior of the system. 1) Agility: An application can specify the agility with which the system must react to changes in the system by specifying the frequency of discovery probes generated by the system. The agility is specified as the interval between successive probes, stated in seconds. 2) Hysteresis: An application can keep the system from reacting to transient changes, not of interest to the application, by specifying a value for hysteresis. The hysteresis is stated in terms of the number of probes for which an application requires the properties of the resources in its context to be consistent before the OFA runtime-environment switches over to the better alternative. This protects the system against thrashing under the fluctuating characteristics of a mobile system. 3) Utility: An application can configure the OFA runtime environment to avoid perfunctory migration of a user-session by specifying the degree of improvement that must result from a session-rebinding before the system moves the user-file to another host. The degree of improvement, k, is specified as a percentage improvement over a previous choice, e.g. 10%.

A user may set these parameters by updating the entries in a special OFA configuration file, OFA_runtime.opts. The OFA runtime environment reads this file at boot-up and configures its discovery and session-rebinding parameters accordingly.

Finally, as a rule-of-thumb, the OFA runtime environment avoids offloading and migration of files when the file is being modifies by a user. To implement this, our system employs a user activity monitor and moves the file only when the user has been idle for *N* seconds (N is configurable by the user).

We are currently using our initial implementation to gather experience with real user studies.

2.3 Data Consistency

When we first set out to design the runtime environment for our system, we planned to use an activity-based (also called an operation-based) file synchronization scheme [7] to synchronize a file offloaded to a remote host with the local copy on the mobile device.

However, an activity-based synchronization scheme involves recording the editing operations on the remote device and playing them back locally to reproduce the modified file. In our case, the "pocket-version" on a handheld mobile device often lacks editing features available on the desktop-variants, making it impractical to reproduce the edits on the handheld device. Therefore, we simply use rsync, a popular application-independent "diff-based" synchronization protocol, which compares a running-hash of non-overlapping chunks of file replicas to exchange modified file chunks over the network. We are, however, currently exploring hybrid approaches that can exploit both activity-based and diff-based synchronization to reduce the size of updates transferred over the network.

By default, the OFA runtime environment provides the standard close-to-open consistency, in which the contents of the remote file are saved after the file is closed for editing (and updates are ensured to appear in a subsequent file-open). Additionally, to protect against device failures, the OFA runtime environment periodically synchronizes the contents of

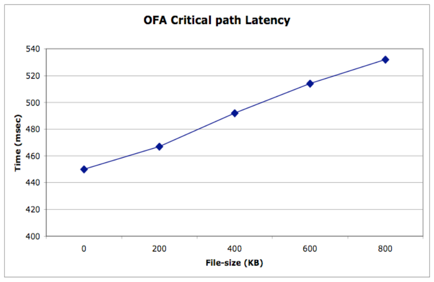

Fig. 2. Latency of file-open with OFA

the remote copy with the local file. The frequency of synchronization, specified in seconds, may be set by the user in the ofa_config.conf configuration file.

While periodic synchronization protects against failure of remote hosts, the OFA runtime environment also provides protection against failure of a mobile device, as well as temporary network disconnection between the mobile client and the remote host. To ensure that the data on a mobile client is eventually made consistent with the latest edited copy, despite a sudden failure of the mobile client, the OFA runtime environment maintains a persistent log of currently open OFA sessions. This log is checked by the system at boot-up and if an open session in the log is found, the OFA runtime environment automatically contacts the corresponding remote host and makes the local-copy up-to-date by running rsync.

3 Implementation and Evaluation

Our current prototype implementation is written in Python version 2.3. The OFA file-manager is written using python bindings for the Gtk+ toolkit. The entire client-side system was written in less than 400 lines of code. User-activity is monitored by interfacing with the OS Window Manager and is currently implemented only for WinCE version 3.0 running on iPAQ PocketPC. We ported rsync (with SSH) to WinCE for file-synchronization. We implemented a simple resource discovery protocol similar to INS [5]; resource descriptions are stored as flat ASCII-encoded attribute-value lists which are periodically discovered over UDP. The suitability of available resources is evaluated by assigning each resource a score based on its attribute-values, akin to SoNS [3]. Our current implementation for WinCE 3.0 does not actually modify a file-association stored in the Windows registry, rather the OFA file-manager maintains its own default file-associations and consults the registry only for accessing the local file-association. The interaction between an OFA client and server is carried over a long-lived TCP connection, while device failures are detected by setting

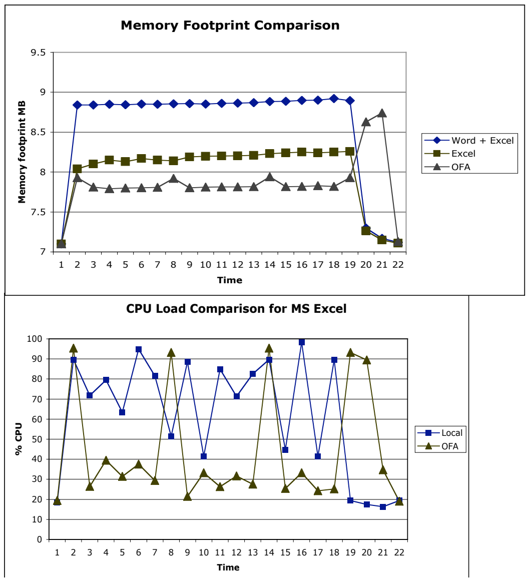

Fig.3.a,b Overhead of OFA runtime environment

the TCP KEEP_ALIVE probing period equal to the agility parameter specified by the user.

3.1 Evaluation

We evaluate the performance of our architecture using two different experiments. First we measure the efficiency of our implementation by measuring the latency introduced by the OFA runtime environment in the critical path of opening a file i.e. time between a user selects a file and it is opened on a remote host. Second, we measure the overhead of our system by comparing its memory footprint and CPU load with pocket versions of MS Word and Excel. Our experiments were conducted on a Compaq iPAQ (Pocket PC), H3800, equipped with a SA1110 ARM processor (clocked at 206 MHz) and embedded with 64 MB RAM and 32 MB Flash ROM. We use a single-slot expansion-sleeve to connect to an Orinoco Silver 802.11b PCMCIA card to the iPAQ. The server is a 2.4 Ghz Intel machine with 256 MB of RAM running WinXP. For our experiments, we stopped all background processes on WinXP. The client and the server were both configured to transfer data at 11 Mb/sec using the same 802.11b access point. Each experiment was run 5 times on WinCE 3.0 and averages were plotted.

Figure 2 shows the time it takes to open a file using the OFA runtime environment. In order to isolate the performance of our system from network and server latency of resource discovery, we use a cached in-memory table of available resources that is consulted by the discovery module when the user “clicks” on a file using the OFA file-manager.

Figure 2 shows a system-initialization time of 450msec (null-sized file-transfer) plus the network latency of copying the file over the network, with an average latency of around 500msec for a 500KB file. This time includes launching MS Excel on the target remote host.

Besides the qualitative improvement afforded by OFA's opportunistic file-offloading mechanism, our system also saves resources on the mobile device. Figure 3a and 3b compare the overhead of running the OFA runtime environment with the resource consumption of MS Word and MS Excel on a Windows CE device. For this comparison, we use a commercial benchmarking tool, Pocket PC Test Suite 2.0 (from Sbp software), to generate simulated screen taps and keyboard events on the mobile device and replicate the same workload on a remote device. The graphs compare the resource consumption of editing a file locally as opposed to making edits remotely with the OFA runtime environment running in the background. The periodic peaks in the OFA resource consumption represent discovery probes, while the memory footprint and CPU load of OFA at the end of the test represents the cost of synchronizing the remote copy with the local file. On average, the OFA runtime environment consumes less than 50% of the CPU time compared to an application like MS Word and has a peak memory footprint of less than 1.2 MB.

4 Related Work

Our work shares elements with a wealth of mobile computing architectures. We share our motivation with network messaging architectures such as INS[5], SoNS[3], LANS[4] and MSOCKS [8], remote execution engines such as 4.2 BSD rexec and Spectra [9] and mobile-agent architectures such as TACOMA [10] and Hive[11]. Conceptually, an Opportunistic-File-Association is akin to a distributed object reference, for instance, as proposed in Obliq [12].

Opportunistic File Associations enable a computing model that is exactly opposite to the one afforded by network file systems such as NFS and Coda [2]. Similarly, Opportunistic-File-Associations complement the popular "synching" applications, which synchronize the data stored on a mobile device with the primary copy on the user's desktop.

5 Summary and Future Work

In this paper, we propose dynamically-assigned, Opportunistic File-Associations (OFAs) as a mechanism for decoupling the storage and handling of user files in a mobile device. A file with an Opportunistic-file-association is not limited to the resource-constrained local application environment on a mobile device. Rather, the runtime environment supporting Opportunistic file associations causes a user file to be opened at the most appropriate device in the environment of the user.

Security is an important area that our current system does not address. Our approach warrants attention to several security concerns. For one, only authenticated devices must be allowed to discover and offload a file to a remote host. Conversely, a client must be able to offload a file only to a trusted server. Threats of malicious code and denial of service attacks also raise serious concerns for offloading files. On shared servers, privacy of previously offloaded files is also an important issue. Eavesdropping of user data when offloading files, as well possible connection-hijacking by malicious servers must be prevented by our system. We are currently working on a comprehensive security architecture for our system.

References

[1] Project Oxygen, MIT CSAIL. http://www.oxygen.csail.mit.edu

[2] J. J. Kistler et al. *Disconnected operation in the coda file system*, 13th ACM Symposium on Operating Systems Principles, 1991.

[3] U. Saif, J. M. Paluska, *Service-oriented Network Sockets*, USENIX Int. Conference on Mobile Systems, Applications and Services, MobiSys 2003.

[4] U. Saif et al, *Practical Experience with Adaptive Service Access*, ACM Mobile Computing and Communication Review (MC2R), Vol 9 Issue 1, 2005

[5] William Adjie-Winoto, et al. *The Design and Implementation of an Intentional Naming System*. Symposium on Operating Systems Principles, December 1999.

[6] S. Czerwinski et al, *An Architecture for a Secure Service Discovery Service*, Proceedings of ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99)

[7] Tae Young Chang et al, *Mimic: raw activity shipping for file synchronization in mobile file systems*, 2nd international conference on Mobile systems, applications, and services, 2004.

[8] D. Maltz and P. Bhagwat. *MSOCKS: An architecture for transport layer mobility*. IEEE INFOCOM '98

[9] J. Flinn, D. Narayanan, and M. Satyanarayanan. *Self-tuned remote execution for pervasive computing*. In HotOS-VIII

[10] D. Johansen et al , *Operating System Support for Mobile Agents*, Fifth IEEE Workshop Hot Topics in Operating Systems, 1995

[11] N. Minar et al. *Hive: Distributed agents for networking things*, ASA/MA'99

[12] L. Cardelli. *A Language with Distributed Scope*. Computing Systems, 8(1):27--59, 1995.

[13] Andrew Wilson, Hubert Pham: *Pointing in Intelligent Environments with the WorldCursor*. INTERACT 2003