OmniStore: A system for ubiquitous personal storage management

Alexandros Karypidis Spyros Lalis

Department of Computer and Communications Engineering
University of Thessaly, Hellas

{karypid,lalis}@inf.uth.gr

Abstract

As personal area networking becomes a reality, the collective management of storage in portable devices such as mobile phones, cameras and music players will grow in importance. The increasing wireless communication capability of such devices makes it possible for them to interact with each other and implement more advanced storage functionality. This paper introduces OmniStore, a system which employs a unified data management approach that integrates portable and backend storage, but also exhibits self-organizing behavior through spontaneous device collaboration.

1 Introduction

Driven by the rapid advances in hardware and communication technologies, the digitally augmented elements around us are increasing, both in number and diversity. Computing logic is embedded into objects such as jewelery [2], wristwatches [15, 9] and clothes [24, 16]. These elements sum up to a wealth of diverse computing and information resources, which future ambient computing systems are expected to discover, manage and exploit in an efficient way, to meet application requirements and user intentions or desires.

Personal area network (PAN) based computing systems comprise of several wearable and portable devices joined in collective operation by sharing resources, also exploiting stationary devices in the surrounding infrastructure in an opportunistic manner. Suitable runtime mechanisms are needed to detect, exploit and manage devices and resources in the PAN, supporting applications in coping with its highly dynamic nature. From a user perspective, this opens the way towards seamless interoperability among mobile and wearable devices, making it possible to build a multi-device personal system on the fly, with minimal explicit (configuration) input.

In this paper we present OmniStore, a framework designed to assist users in managing storage across multiple devices. The design of OmniStore encompasses both infrastructure and portable storage, supporting automated and asynchronous data flow between PAN devices and servers. Furthermore, distributed storage access is provided in the PAN, allowing applications on any device to flexibly access storage in neighboring devices. File transfers are also performed behind the scenes to distribute storage load among devices or to ensure availability of important data through replication inside the PAN. As a result OmniStore exhibits self-organizing behavior and automates cumbersome tasks which users often face when dealing with storage on personal computing systems.

The rest of the paper is structured as follows. Section 2 provides an overview of OmniStore's design and the rationale behind it. Section 3 presents and discusses various aspects of our implementation. Related work is reviewed in Section 4, concluding with a summary of our contributions in Section 5.

2 OmniStore design

In designing OmniStore we reviewed typical roles of portable devices with storage, identifying three primary modes of operation: data producers, data consumers and data couriers. These are discussed in the following, giving indicative examples which provide a better perspective.

Producer devices create files. A common example is a digital camera, which is used to take photographs. The files produced are (later) transfered to a personal computer to release storage on the camera. Similarly, a digital voice recorder is used to produce sound files which serve as audible memos that may be played back later. A mobile phone with call-recording capability is often used in the same way.

Consumers are devices on which files are being actually used. A music player is a typical representative of this class. Applications for managing music collections allow users to purchase music from Internet stores and create playlists that are placed on the music player. Another type of data consumers that employ a similar model are portable gaming devices.

Courier devices transfer files. They are employed when communication between a producer and a consumer is not physically possible or just to increase the availability of a file. Any device with storage can be used in this manner. For instance, prior to going on a conference trip we usually place a copy of our presentation on our PDA (or USB stick). This is because even if the presentation were accessible through the Internet, a portable can be quite useful in case of rare (but always possible) network and server failures or security restrictions that may hinder file downloading.

An important observation is that portable devices, though autonomous in operation, function in conjunction with a greater and highly available personal storage service. The necessity of such a service arises from the fact that portable devices have limited storage capacity and can be easily forgotten, misplaced or lost. Technically, the personal storage service can be implemented in a distributed way, e.g. employing multiple servers of an Internet Service Provider, or in a centralized fashion, e.g. on a home server appliance with a permanent connection to the Internet.

The role of portable devices can be further refined in the context of such a personal storage service. More specifically, each producer device acts as a write-through file cache towards the personal storage service. Data consumer devices acquire files from the personal storage service and cache them in order to be used by local applications. Of course, the existence of the personal storage service does not prohibit portable devices from interacting directly with each other. In fact a data courier device may intervene between the personal storage service and producer or consumer devices, facilitating the corresponding file transfers.

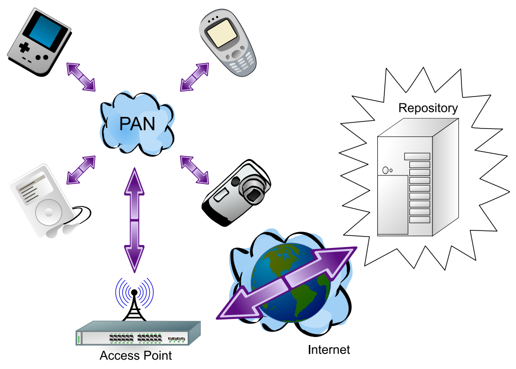

Figure 1. Elements of the OmniStore design

We have crafted our design around this model, namely: *OmniStore combines an infrastructure-based personal storage service and portable storage devices, with the latter acting as intermittently connected caches for the former.* Our approach is illustrated in Figure 1. In OmniStore, all files generated on portable devices are forwarded to the personal storage service, referred to as the *repository*. It is also possible to select files from the repository to be cached on portable devices. Moreover, OmniStore introduces collaborative behavior among portable devices in a PAN to achieve the abstraction of a *single* collective portable cache. Towards this goal, transparent remote file access and automated file off-loading is supported between devices, which is used to increase availability and reliability, but also to distribute storage load as required.

3 Implementation

OmniStore is built on top of our runtime for PAN-based computing system, which provides the necessary services for ad-hoc interaction among portable devices. The runtime is written in the Java language, uses only classes from the `java.lang` package and is just over 200KB in size. Thus it may easily be used with J2ME installations for embedded devices.

Due to space limitations we do not discuss the implementation of the core runtime here, but refer the reader to our previous publications. A brief overview of our PAN-based computing vision and runtime functionality is given in [11], basic service discovery and point-to-point communication facilities are discussed in [12], heuristic peer selection is presented in [8], and the maintenance of a shared contextual perception in a PAN is discussed in [7]. In the rest of this paper we will casually refer to these facilities without giving further details about their implementation.

3.1 System overview

The OmniStore system comprises several components. Portable storage devices run the OmniStore *device daemon*, a process that is responsible for performing the necessary communication with other portable devices, as well as with the repository. Applications running on portable devices use the OmniStore *device library* API for file access; the library also performs behind the scenes interaction with the device daemon. The functionality of the repository is implemented via the *repository daemon* process that services requests sent from remote device daemons. It also provides a web interface that can be used to perform various file browsing and management operations via fully-connected infrastructure applications. Finally, the *Internet access dameon* acts as a network gateway, tunnelling connections from portable devices to services (such as the repository daemon) on the Internet.

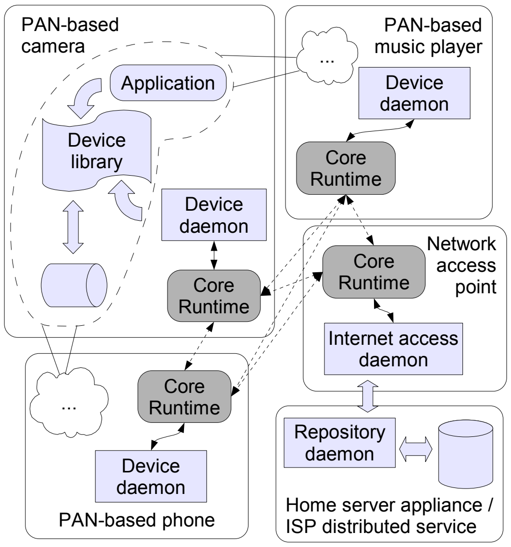

Figure 2. OmniStore architecture

The laboratory setup used to test our system is as follows: The repository daemon is installed on a server with a wired connection to the Internet. The Internet access daemon is installed on several PCs equipped with both wired and wireless adapters, acting as network access points. PDAs are used as portable storage devices, running different applications that simulate more purposeful devices such as music players, digital cameras and mobile phones. Figure 2 depicts a typical configuration that includes the repository, a single access point and a PAN with three portable devices.

3.2 Naming and configuration

Our PAN-based computing framework provides naming and configuration services via the *device registry*, an infrastructure directory service for user devices. Although the physical location of this service is irrelevant, one can imagine this being installed together with the repository service. The registry is identified by a unique DNS name[1], e.g. `userHomeServer.someISP.org`. The registry's name is the root of the user's personal namespace; thus it is sufficient for all other identifiers to be unique in context of this namespace.

Newly acquired personal devices must be registered with the device registry in order for them to be used as a part of the personal system. During registration, device-specific credentials are produced which allow the device to authenticate itself to infrastructure services (including the repository daemon), along with a certificate that can be used by devices to prove (to other portable devices) their participation in the user's domain. Furthemore, a unique (within the context of the registry's namespace) 32-bit identifier is issued for the device. The combination of registry name and device identifier constitutes a globally unique identifier for the device.

Once a device is registered, various device specific configuration parameters can be stored on the registry. Some parameters can be a priori known and widely accessible so that they can be used by all applications. One such configuration parameter is the *mnemonic name* which can be set to a human readable name so

[1]This is sufficient for our purpose in spite of the numerous problems and shortcomings of DNS; see [1, 25].

that applications are able to display it to the user, instead of the device's unique identifier. Other parameters may be reserved for special system use. For example, OmniStore uses some configuration parameters to trigger and control data management activities; see Sections 3.4 and 3.7.

3.3 File organization

OmniStore issues globally-unique identifiers for every file. To achieve this, each device maintains a 32-bit local seed that is incremented with every file creation. This value, combined with the identifier of the local device, constitutes a unique file identifier across all devices in the user's realm. The corresponding globally unique file identifier is then produced by prefixing the name of the registry. Figure 3 gives an example of a mobile phone containing two files: a photograph that was created on the user's camera (but was since transferred to the phone) and a locally created phonecall recording.

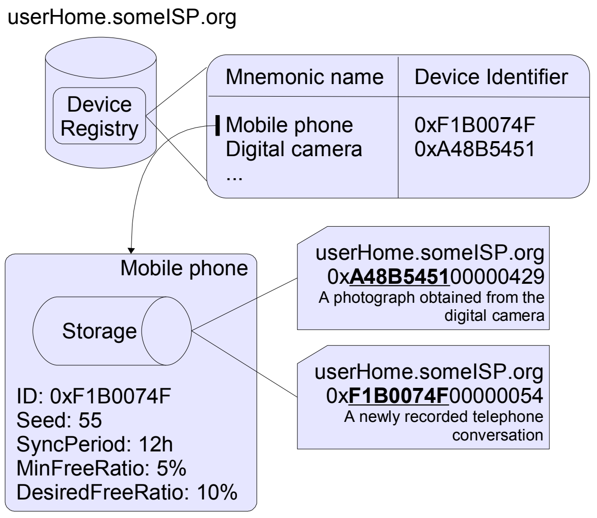

Figure 3. Various elements labeled using our naming scheme

File identifiers are of course hard to memorize and therefore quite inconvenient for people. To support flexible organization and browsing, OmniStore allows files to be annotated with extra (semantic) information which can be used to group or sort files in a flexible way. This approach, presented in [3, 4, 14], is increasingly gaining in popularity. In OmniStore, annotations are defined as key-value pairs whose interpretation is left to applications. The storage library allows applications to create, review and edit annotations. There are some predefined annotations, listed in Table 1, which are reserved for system use. Some of them, e.g. `mnemonic-name` and `replicate-count`, can be accessed by any application. The rest are reserved for system use only. Also, as discussed in Section 3.9, OmniStore may automatically generate an open number and type of annotations for newly created files.

Key	*Values and use*
dirty	Flag indicating whether the file exists in the repository or not.
mnemonic-name	A human-readable name for referring to the file.
derived-from	The value contains a file identifier indicating that the annotated file is a subsequent version of that file.
identical-content	Indicates that this file's content is identical to that of an ancestor (Section 3.8).
avail-start	Timestamp indicating the start of the availability period during which the file should reside on the local device (Section 3.5).
avail-end	Timestamp indicating the end of the availability period during which the file should reside on the local device (Section 3.5).
replicate-count	Indicates that this file must be replicated a certain number of times (Section 3.7).

Table 1. Most relevant system-defined annotations

OmniStore employs the write-once-read-many (WORM) model, as introduced in the Cedar system [20, 5]. In other words, a file remains *immutable* once created. To avoid changing a file when it is opened in read-write mode, a new file is created by copying the contents of the original and the returned file handle refers to the copy. The copy also inherits the annotations of the original file. To maintain file history, OmniStore also attaches a `derived-from` annotation with its value being the identifier of the original file.

We note that files are stored in a single directory in a *flat* manner, both on portables as well as in the repository, using file identifiers as file-system names. The globally unique nature of file identifiers makes name clashes impossible: files with the same name are guaranteed to be identical copies.

3.4 Automated backup and deep archival

Device daemons maintain a backup queue containing references to the files that need to be transfered to the repository. When a new file is created, it is automatically added to the local daemon backup queue. In addition, a `dirty` annotation with a value of `true` (see Table 1) is attached to the new file. The device daemon periodically tries to contact the repository daemon (the value of the retry period is specified via the `syncPeriod` device configuration parameter). As soon as a connection is established with the repository, the backup queue entries start being processed in sequence. For each entry, the device daemon sends the file's identifier, annotations and size to the repository[2], which records this information. The repository replies with a list of offset-length pairs indicating the file fragments missing from the repository, and the device daemon sends the corresponding parts. This interaction is repeated until the repository replies with an empty list. The device daemon then sets the file's `dirty` annotation to `false`, removes it from the backup queue and proceeds with the next entry. Figure 4 illustrates an indicative interaction scenario for a single file transfer.

The backup protocol is designed to support progressive file backup over intermittent connections. In case of communication breakdown and re-establishment, data transfer will resume starting from the last fragment that was successfully received by the repository. If a device attempts to backup a file that has already been uploaded to the repository, perhaps by another device[3], it will be prompted to continue with its next backup entry. Backup can also be performed in a *collaborative* fashion, in parallel or sequence, by several devices holding copies of the same file. As an example, Figure 5 shows a scenario where backup is initiated by device A but is then completed via device B.

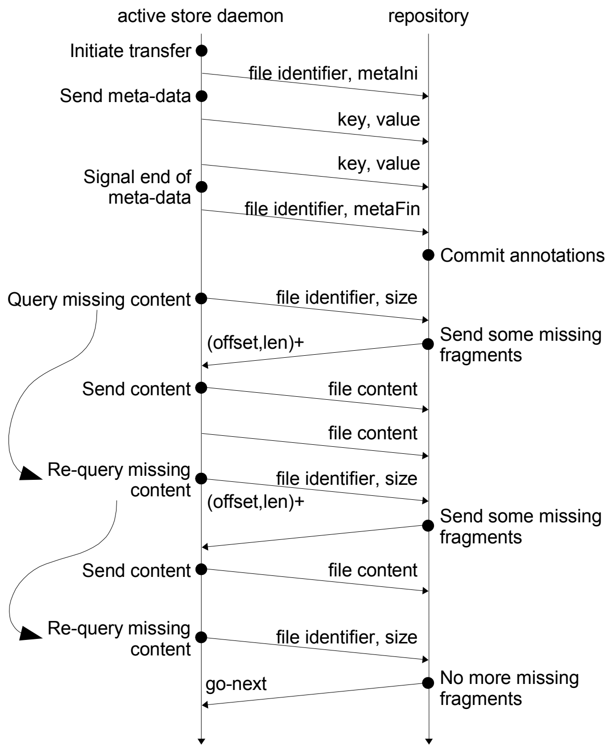

Figure 4. OmniStore backup protocol

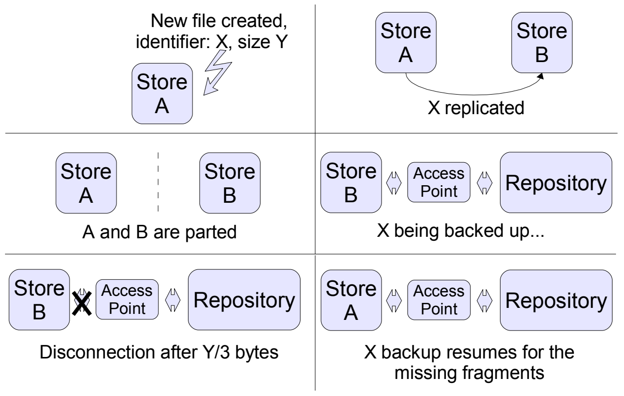

Figure 5. Collaborative backup

As a result of the backup process and the use of the WORM model, all versions of all files ever created are eventually collected in the repository. This achieves the *deep archival* [14] property, which is considered valuable for data organization purposes; it is also be-

[2]Some system annotations such as `dirty` and `replicate-count` are device-local and are thus not sent to the repository.

[3]File copies may be automatically created on several other devices via the system's mechanisms (see Section 3.7).

coming practically feasible in view of the ever increasing available storage capacity at ever falling prices.

3.5 Caching of repository files

Besides receiving files from portables, the repository also provides a service for sending files to portables. We refer to this as *push-caching*. Push-caching is activated by submitting a request to the repository, with the desired file and target device identifiers. When the daemon of the target device contacts the repository daemon, it is informed about the files that need to be transfered locally, if any, and adds corresponding entries to its fetch queue. The fetch queue of the device daemon is processed in analogy to the backup queue, as described in Section 3.4, following a similar protocol to download files from the repository.

Push-caching requests can be submitted with an *availability period* during which the file should reside on the target device. This allows for conscious scheduling according to the user's plans, but also for increased flexibility at the runtime level since it becomes possible to defer a data transfer if the specified time frame is far into the future. Push-caching requests can also be submitted with the *live-update* option enabled, indicating that the target device should receive the *latest version* of the specified file identifier. To achieve this, the repository uses the *derived-from* annotation to find the file's most recent descendants and sends the corresponding identifiers to the target device daemon, as soon as it is contacted by it. Live-update requests with an availability period are automatically removed when they expire. Else, the target device is updated indefinitely until the push-caching request is explicitly removed.

It is important to stress the flexibility of the push-caching facility. Firstly, there is no need for the target device to be physically present, remotely reachable or even turned on when the request is issued. Secondly, by specifying an availability period file transfers can be performed well in advance of the expected file usage, thereby considerably reducing the chances of a file being unavailable when it is expected to be found on a given device. Thirdly, when using the live-update option, the issued request essentially concerns the future versions of a given file, i.e. files that will be derived from it. The request may nevertheless be issued by the user at a convenient point in time, rather than waiting for the latest version to be actually produced / committed.

As a test tool, we have created a web interface that allows the user to issue such requests to the repository daemon from any networked terminal with a web browser. We note however that the push-caching service may be invoked from any application, whether running on a portable device or on an infrastructure system. File transfers may for example be scheduled by the playlist manager of a music player, or even an advanced / intelligent subsystem that infers the user's schedule based on calendar and agenda information.

3.6 File access

The OmniStore library provides a fairly typical API, allowing applications to open, read and write files as in other network file-systems. An exception is that data locality is required for modifying files. Opening a remote file in read-write mode results in a copy being created (and written to) on the local device, using a new file identifier. Read-only access is allowed directly from remote devices. Since files are immutable, remote file handles can also change the source from which data is retrieved to enhance availability. In other words, if a file is opened from device A and the connection is broken, file access / transfer may be transparently resumed / redirected using another device B that holds a copy.

In order to open a file its identifier must be provided. File identifiers may be retrieved by looking up files via their annotations, including their `mnemonic-name`. For this purpose, the OmniStore library allows applications to create and register so-called *annotation lookup tasks*. Search expressions can be based on specific keys or generic terms. As an example, Listing 1 shows a code excerpt that creates a task for looking up files that satisfy a combination of two criteria: they must be annotated with a 'file type' key that has the value 'photograph'; they must have an annotation with its key or value containing the term 'cafeteria' (this matches annotations such as ['location', 'Grapa cafeteria'] or ['cafeteria', 'Jam']).

Lookup tasks are constantly being populated with matches based on the files that are available in local *as well as* remote devices in the PAN. The device daemon

```
LookupTask lt = storage.createTask();
lt.searchKey ("file␣type",
    "photograph");
lt.searchTerm ("cafeteria");
lt.addMatchListener(this);
lt.register();
```

Listing 1. A sample Component object

is notified about the arrival and departure of nearby devices by the core runtime. Newly discovered devices are inquired about files matching the search terms of the locally registered lookup tasks, whose contents are updated accordingly. When a device leaves the PAN, the matches that have been produced by it are removed. Applications may introduce reactive behavior by supplying a notification listener to be up-called when the result set of a lookup task changes.

3.7 Storage reclaim and replication

OmniStore automatically creates free space on devices whose storage is filling up. Storage space reclamation is driven by the `minFreeRatio` (MFR) and `desiredFreeRatio` (DFR) device configuration parameters. When free space drops below the MFR, the device daemon attempts to remove local files that are not needed. Candidates for deletion are the least recently accessed files that (i) have been successfully backed up in the repository and (ii) are not expected to be available on the local device in the near future. The former can be determined by inspecting the `dirty` annotation whereas the latter can be determined by inspecting the `avail-start` and `avail-end` annotations (which are created by the storage daemon as it fetches files following push-caching requests). Garbage collection stops when the DFR is reached.

Though unlikely, it is possible that the DFR cannot be met given the above garbage collection restrictions. As a last-resort option, the device daemon may violate condition (i), but it must first create a copy on another device which will assume responsibility for backing up the file to the repository. When such a device is found, the file is transferred between the two device daemons by employing a protocol that is similar to the repository backup protocol, allowing for intermittent connectivity. The file is not deleted locally unless it has been completely off-loaded to the other device. The transfer can be unilaterally aborted by both parties at any point in time, in which case the receiving device is free to delete the partly transferred file to reclaim storage; which will occur naturally when garbage-collection kicks in.

A related activity which may occur among portable devices is file replication. This is driven via the `replicate-count` annotation of a file. By setting this value to a number greater than zero, the device daemon will attempt to create equally many copies of this file on other portable devices. The transfer protocol employed is the same as for file off-loading, except for the file not being deleted locally. Applications may set this annotation when they decide that a file is of great importance. In fact, the OmniStore device daemon adds this annotation to files retrieved following push-caching requests that have an availability period specified. This is done in order to decrease the probability of file unavailability if the user happens to not have this device around during that period.

3.8 Annotation management

As discussed in Section 3.3, files may be decorated with meta-information by attaching annotations. The aim is to associate semantic information with them, so that versatile lookup and browsing schemes (such as virtual directory hierarchies [3, 4, 27]) can be implemented. In short, annotations are used to *qualitatively* describe a file's content for organizational purposes.

File annotations typically have stable values. Compared to the addition of annotations, their editing and removal is quite uncommon. This applies even more for portable devices since their limited U/I resources make the interactive inspection and updating of annotations almost impossible. Nevertheless, given that the OmniStore library provides such operations, it is in principle possible for applications running on portable devices to alter or remove annotations. Thus, rudimentary synchronization support is required.

In accordance with the WORM access model, editing annotations creates a new file revision. We enforce this through the OmniStore library, enabling annotation modification functions only for files opened in read-write mode. Annotations may only be edited while the file is open, and are frozen to their current values as soon as the file is closed. The newly created file is added to the backup queue as usual.

To efficiently manage the case where *only* the annotations of an opened file are modified, file handles contain a flag which indicates whether the `write()` function was called. If a file is closed with unmodified content, an annotation with key `identical-content` is added to the newly created file, which records the fact that it has the same contents as one of its ancestors (the value of this annotation is the file identifier of that ancestor). Specifically, if the original file has an `identical-content` annotation itself, then its value is used for the new file as well; else, the original file's identifier is used as the value for the `identical-content` annotation. This information is exploited in all file transfer scenarios (backup, push-cashing, off-loading, replication) to avoid the transmission and storage of identical file contents under different file identifiers.

Our approach to annotation management re-uses our basic mechanisms, adding minimal load to the portables. Version branches created when editing annotations concurrently on two different devices are properly recorded. If needed, they can be resolved at a later point in time, e.g. via a special tool that inspects the contents of the repository (user input may still be required though to resolve conflicting changes).

3.9 Automated annotation

A significant issue in semantically-organized systems is the process of adding annotations to files. This is a cumbersome task which is equivalent to that of creating directory hierarchies in traditional file-systems and moving files around, in order to organize them thematically. Some systems [3, 22] use agents which perform data mining to populate files with annotations (for example searching for the `Subject:` line in e-mails). More recently, the use of application context [21] was suggested as a source of file annotations (for example, annotating an image saved by the browser after a search on the internet for 'penguin', with the search expression that was used to locate the image).

The OmniStore system is completely open in this respect, allowing higher-level mechanisms and applications to add and exploit annotations in a flexible fashion. For example, it is possible that a photograph is annotated with data such as the temperature, lighting conditions and GPS coordinates at the time and place were it was taken. Notably, a context recording and diffusion mechanism for the maintenance of a shared contextual perception among devices in a PAN has already been investigated in conjunction with OmniStore in [7]. This makes it possible to flexibly generate annotations for files created on a portable device, based on the sum of the context information provided by the devices in the PAN and without requiring explicit user input.

Our approach in context provision is similar to that of semantic annotations and uses key-value pairs. Consider for example an airport with location-beaconing devices informing portable devices of their location via key-value pairs such as ['location', 'Athens International Airport'], ['City', 'Athens'] and ['Country', 'Greece']. Also consider a mobile phone that is able to characterize a call via key-value pairs such as ['call', 'incoming'] and ['party', 'John Doe']. Assume that the user decides to record part of the conversation as a voice-memo. OmniStore will then automatically annotate the file using the key-value pairs of the available context information. When the user wishes to access the file at a later point in time, she may look it up using terms such as 'Athens', 'airport' or 'John'. A detailed description of this functionality can be found in [7].

3.10 Educated peer selection

PAN-based systems are often required to select among several peers in order to perform some operation. Intelligent selection of a service when multiple possibilities exist can lead to increased performance and the avoidance of inconvenient situations. More specifically, in the case of OmniStore, a portable device that wishes to off-load or replicate a file (see Section 3.7) may detect two or more candidate recipient devices. Although making a random choice is one way to solve the dilemma, in many cases this could significantly degrade file availability.

Our approach and runtime support for making a more educated guess is presented in [8]. The main idea is to select a partner device based on its co-location pattern, i.e. statistics regarding how often it is found close to the local device. OmniStore exploits this facility to select devices which are statistically likely to remain in the vicinity of the device initiating the transfer. This decreases the probability of the transfer being interrupted frequently or permanently. It also increases the probability of the transferred file remaining within range of the sender device. For instance, when a device off-loads a file to reclaim storage, it will most likely still be able to access the file remotely in the future. When a device replicates a file to increase its availability, the file will be copied on a device which is usually close to it, hence the latter can act as a more effective backup of the former.

4 Related work

Portable storage does have advantages over distributed file systems. For this reason, its popularity has not been diminished, in spite of the increasing level of network coverage. Portable storage offers guaranteed performance; in contrast, distributed file system performance is subject to network congestion. Availability is also guaranteed, provided the storage medium does not fail. Distributed file systems – in addition to server failure – are also subject to other types of failure (such as network outage, operator errors, etc). An overview of the advantages and disadvantages of the two fronts can be found in [23], which justifies the place of portable storage in upcoming ubiquitous computing environments.

Based upon the utility of portable storage, the concept of the 'ultimate' portable storage device was investigated in [26]. The idea is that all user data resides in the personal server, a single portable storage device with ad-hoc networking capability and no U/I elements. The user accesses the data through terminals in the environment which form ad-hoc connections to the personal server. This approach is interesting as it takes the active store concept to the extreme, although admittedly one is unlikely to discard of all backend storage, if only to avoid disaster scenarios in which the personal server fails and all user data is lost. Such a device nicely fits into our model as a more capable portable store that is carried by the user more often than others. In fact, we have implemented a similar device, called the electronic wallet, which was used as the main data and application store of our first prototype system [12].

Rather than centering the ultimate storage solution around one portable device, the other end of the spectrum is the unified management of both portable and backend storage in which portable storage acts merely as a cache. This is also the approach taken by in OmniStore. A similar concept can be found in [23], where the 'lookaside caching' technique is presented. Lookaside caching allows updating the files on a portable storage medium when it is mounted, by means of a hash function (the authors use SHA-1) which triggers updating of files whose hash has changed on the server. While this work was devised with passive stores (non-network-capable storage devices) in mind, it is equally applicable to PAN-based stores: a dismounted passive store corresponds to a device that cannot communicate with the repository. The provided functionality is similar to push-caching technique with live-update enabled on all files. However, by using the WORM approach [14] for deep archival, combined with our naming scheme, we are able to detect different file versions through simple comparison of file identifiers.

Moving on to infrastructure-based approaches, Coda [19] is the most well-known system addressing mobile computing. It uses optimistic caching to replicate the working set of user files on laptops and keep them in sync with the server. The UbiData [6] system builds upon Coda to address the existence of resource-limited clients such as PDAs. It supports transcoding of data in combination with a system for 'format-independent change detection and propagation', which allows consolidating changes made to transcoded versions using different applications. This is aimed to enable data editing via applications which employ different formats, a common case in stripped-down PDA versions of desktop applications. Both Coda and UbiData mainly address relatively rich clients such as laptops and PDAs, which have much in common with the personal computer model. Our design targets less capable, specific-purpose devices and assumes that users will want to carry and use several such devices at the same time. We also introduce significant collaborative functionality among portable devices.

An improvement over a centralized repository is the use of a distributed backend storage service. The Roam system [17] which uses peer-to-peer communication and can perform synchronization among any two replicas seems well-suited for deploying such an infrastructure. Another equally sophisticated system is OceanStore [10], which can further exploit untrusted servers for storing information. Needless to say, OmniStore's repository would benefit from a distributed approach, in which multiple backend servers are used to hold user data.

The approach of treating files as immutable objects was first introduced in Cedar [20, 5]. Deep archival offers several advantages and its use has been employed by systems such as the Elephant [18] file system and more recently Sedar [14]. The later is closer to our design as it combines both deep archival and semantic organization. Again, these systems do not address mobile devices. We point out that since deep archival keeps all revisions of all files in the repository, data reconciliation may occur at a higher level using approaches such as [13] to generate merged copies.

Semantic annotations have been the target of research for a while [3, 4] and researchers seem to agree that they are a more flexible and powerful way of managing files. Recently, their use in combination with deep archival [27] was suggested to further simplify storage management. The core issue now is the automatic generation of annotations [21] in order to relieve users from manual entry. In [7] we present a method of automatically annotating files generated on portable active stores using context information (e.g. sensor data) of various devices which are present in a PAN. A significant amount of meta-information can be generated using our technique with absolutely no user input.

In terms of meta-data management, our system shares some analogy with the Roma [22] personal meta-data service. Roma uses a server to store meta-information regarding user files. It operates at higher level than OmniStore, using URIs to refer to files for which it holds meta-data. However, Roma does not deal with the files themselves and makes no provisions for automating storage management operations.

5 Conclusion

The essence of ubiquitous computing lies in enabling people to attend to their tasks, without diverting their attention to the details of making technology work. In other words, one should be assisted in achieving her goals, instead of being distracted by the inner workings of computing systems.

In its current form, OmniStore addresses many aspects of personal storage management in multi-device and mobile computing environments. It supports flexible data placement, taking into account the various devices that can be used to generate, store and consume information. Applications (and the user) need not care about the actual location of data, nor manually move and synchronize data between devices. Such operations are performed asynchronously and in an incremental manner, behind the scenes. Moreover, they do not require user supervision, not even user input in some cases. We thus believe that it is a step forward in applying the ubiquitous computing vision to personal data management.

References

[1] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 343–352, New York, NY, USA, 2004. ACM Press.

[2] M. Gandy, T. Starner, J. Auxier, and D. Ashbrook. The gesture pendant: A self-illuminating, wearable, infrared computer vision system for home automation control and medical monitoring. In *Proceedings of the 4th IEEE Internation Symposium on Wearable Computing*, pages 87–94, October 2000.

[3] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 16–25. ACM Press, 1991.

[4] B. Gopal and U. Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the third symposium on Operating systems design and implementation (OSDI '99)*, pages 265–278. USENIX Association, 1999.

[5] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 155–162, New York, NY, USA, 1987. ACM Press.

[6] A. Helal and J. Hammer. Ubidata: requirements and architecture for ubiquitous data access. *SIGMOD Rec.*, 33(4):71–76, 2004.

[7] A. Karypidis and S. Lalis. Automated context aggregation and file annotation for PAN-based computing.

Personal Ubiquitous Comput., Accepted for publication, DOI=10.1007/s00779-005-0061-4.

[8] A. Karypidis and S. Lalis. Exploiting co-location history for efficient service selection in ubiquitous computing systems. In *2nd International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2005): Networking and Services*, pages 202–209. IEEE Computer Society Press, July 2005.

[9] A. Kirkeby, R. Zacho, J. D. Mackinlay, and P. Zellweger. TrekTrack: A round wristwatch interface for SMS authoring. In *Proceedings of the Third International Conference on Ubiquitous Computing*, volume 2201 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2001.

[10] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM Press.

[11] S. Lalis, A. Karypidis, and A. Savidis. Ad-hoc composition in wearable and mobile computing. *Commun. ACM*, 48(3):67–68, 2005.

[12] S. Lalis, A. Karypidis, A. Savidis, and C. Stephanidis. Runtime support for a dynamically composable and adaptive wearable system. In *Proceedings of the 7th IEEE Internation Symposium on Wearable Computing*, pages 18–21, October 2003.

[13] T. Lindholm. Xml three-way merge as a reconciliation engine for mobile data. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 93–97, New York, NY, USA, 2003. ACM Press.

[14] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. In *Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)*, pages 115–121. IEEE Computer Society, May 2003.

[15] C. Narayanaswami, N. Kamijoh, M. Raghunath, T. Inoue, T. Cipolla, J. Sanford, E. Schlig, S. Venkiteswaran, D. Guniguntala, V. Kulkarni, and K. Yamazaki. IBM's Linux watch: The challenge of miniaturization. *IEEE Computer*, 35(1):33–41, Jan. 2002.

[16] J. Rantanen, J. Impio, T. Karinsalo, A. Reho, M. Malmivaara, M. Tasanen, and J. Vanhala. Smart clothing prototype for the arctic environment. *Personal and Ubiquitous Computing*, 6(1):3–16, 2002.

[17] D. Ratner, P. Reiher, and G. J. Popek. Roam: a scalable replication system for mobility. *Mob. Netw. Appl.*, 9(5):537–544, 2004.

[18] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles (SOSP '99)*, pages 110–123. ACM Press, 1999.

[19] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems (TOCS)*, 20(2):85–124, 2002.

[20] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. *SIGOPS Oper. Syst. Rev.*, 19(5):25–34, 1985.

[21] C. A. N. Soules and G. R. Ganger. Why can't I find my files? New methods for automating attribute assignment. In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*. USENIX Association, May 2003.

[22] E. Swierk, E. Kiciman, N. C. Williams, T. Fukushima, H. Yoshida, V. Laviano, and M. Baker. The Roma personal metadata service. *Mobile Networks and Applications*, 7(5):407–418, 2002.

[23] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proceedings of the 3rd Conference on File and Storage Technologies (FAST '04)*, pages 227–238, San Francisco, CA, March 31 - April 2 2004. USENIX.

[24] A. Toney, L. Dune, B. H. Thomas, and S. P. Ashdown. A Shoulder Pad Insert Vibrotactile Display. In *Proceedings of the 7th IEEE Internation Symposium on Wearable Computing*, pages 35–44, October 2003.

[25] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from dns. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design (NSDI '04)*, San Francisco, CA, March 2004. USENIX.

[26] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light. The Personal Server: Changing the Way We Think About Ubiquitous Computing. In *Proceedings of the 4th International Conference on Ubiquitous Computing*, 2002.

[27] Z. Xu, M. Karlsson, C. Tang, and C. Karamanolis. Towards a semantic-aware file store. In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*. USENIX Association, May 2003.