ZZFS: A hybrid device and cloud file system for spontaneous users

Michelle L. Mazurek[*], Eno Thereska[+], Dinan Gunawardena[+], Richard Harper[+], and James Scott[+]

[*]Carnegie Mellon University, Pittsburgh, PA
[+]Microsoft Research, Cambridge, UK

Abstract

Good execution of data placement, caching and consistency policies across a user's personal devices has always been hard. Unpredictable networks, capricious user behavior with leaving devices on or off and non-uniform energy-saving policies constantly interfere with the good intentions of a storage system's policies. This paper's contribution is to better manage these inherent uncertainties. We do so primarily by building a low-power communication channel that is available even when a device is off. This channel is mainly made possible by a novel network interface card that is carefully placed under the control of storage system protocols.

The design space can benefit existing placement policies (e.g., Cimbiosys [21], Perspective [23], Anzere [22]). It also allows for interesting new ones. We build a file system called ZZFS around a particular set of policies motivated by user studies. Its policies cater to users who interact with the file system in an *ad hoc* way — spontaneously and without pre-planning.

1 Introduction

Much work has been done in developing appropriate data placement, caching and consistency policies in the "home/personal/non-enterprise" space (e.g., see [8, 10, 16, 19, 20, 21, 22, 23, 24, 26, 28]). Good policies are crucial in maintaining good performance, reliability and availability. Unfortunately, there are many barriers that make the execution of such policies far from automatic. These barriers often stem from the unpredictability of everyday life, reflected in variable network resources, devices being off or dormant at inconvenient times, and users' time and priority given to data management.

Consider two mundane examples (Section 2 has more): In the first example, a busy mom desires to show a friend in the mall a photo that happens to be on the home computer. That same person might wish to access her personal medical file (that she does not trust the cloud for storing) from the beach while on holidays later in the week. In all likelihood she will find the above tasks impossible given that her home computer is most likely dormant or off, and she has not had time to specify any particular data replication policy among the computer and the smartphone, or hoarded the files beforehand.

The second example illustrates a consistency problem and is taken from Live Mesh's [14] mailing list. Many technology-savvy users experienced frequent conflicts with music files. A single user would listen to music on device A, then later listen to the same music on device B while A was turned off (the files were kept in peer-to-peer sync between A and B because the user did not have enough space on the cloud to store all files). Because the particular music player software updated song metadata (like play count and rating), it turns out that this is not a read-only workload. As a result, the syncing generated conflicts requiring manual resolution whenever the user switched devices. It is unfortunate that even in the absence of true multi-user concurrency, a single user can still get an inconsistent view of the system.

This paper's main contribution is to build a low-power, always-on communication channel that is available even when a device is off. The hypothesis is that this channel reduces the likelihood that a device is unreachable and thus helps the execution of data placement and consistency policies. We build this channel using new hardware and storage system protocols.

On the hardware front, we incorporate a novel network interface card (NIC) in the design of the overall storage system (Section 3.1). The NIC maintains device network access with negligible energy consumption even when the device is dormant. The NIC is able to rapidly turn on the main device if needed. The ability to turn on the main device can be thought of as Wake-on-Lan(WoL) [11] "on steroids," in that the NIC operates through any firewalls or NAT boxes, does not need to know the MAC address of the dormant device, and handles mobility across subnets. The NIC also exports to our storage system a small on-board flash storage. While the hardware part of the NIC is not a contribution of this paper, we build the storage system software around it.

We design the I/O communication channel on top of the NIC by leveraging several technical building blocks. These are not new individually, but, as we discovered, work well together to lead to a usable system. In particular, we use data placement protocols based on version histories for ensuring consistency (Section 3.3); I/O offloading [15, 29] is used to mask any performance latencies of turning on a device on a write request by using the NIC's flash storage as a versioned log/journal (Section 3.3); and users get a device-transparent view of the namespace with the metadata by default residing on the cloud. Metadata can also reside on any device with the always-on channel implemented (Section 3.2).

Fundamentally, our approach makes good use of any always-on resources, if available (such as the cloud or a home server), but also actively augments the number of always-on resources by turning any personal device with the new network interface card into an always-on resource. Perhaps subtly, however, it turns out that having a few extra always-on resources allows for interesting data placement policies that were not possible before. We explore these through building a file system called ZZFS. We chose to implement a unique set of data placement and consistency policies that cater mostly to spontaneous users (Section 4). These policies were partially influenced by qualitative user research. However, other policies (e.g., Cimbiosys [21], Perspective [23], Anzere [22]) would equally benefit.

2 Background on the problem

Users often have access to a set of devices with storage capabilities, such as desktops, laptops, tablets, smartphones and data center/cloud storage. *Data placement* policies revolve around deciding which user's data or files go onto which device. Often, a data placement policy indicates that the same file should be placed on multiple devices (e.g., for better reliability, availability and performance from caching). *Consistency* policies revolve around ways of keeping the multiple file replicas in sync as to provide the abstraction of a single file to users. We illustrate problems related to the execution of these policies through three simple examples, that reflect policies taken from some recent related work.

Example 1: Two replicas of a file: This example defines the terminology and thus is slightly longer than the subsequent two. Systems like Perspective [23], Cimbiosys [21] and Anzere [22], allow a photographer to say "keep all my photos replicated on my work machine and tablet." Imagine a user U and a photo file F. It is very likely that when U edits F from the work machine, the tablet is dormant so the changes do not immediately propagate to the tablet. Typical implementations of this policy make use of a transaction log L that keeps track of the changes U makes on the work machine. The log is later replayed on the tablet to maintain consistency.

When the photographer later on moves to work on the tablet, the log will still be on the now-dormant work machine. Thus, the tablet is not able to replay the log. The user has two options, neither which leads to great satisfaction with the system: option 1 is for the user to manually turn on the work machine and wait until all the data is consistent. This option is implicitly assumed in Perspective, for example. Option 1 may be out of reach for non tech-savvy users who just want to get on with their work and do not understand they have to wait ("for how long?") for consistency to catch up.

Option 2 is to continue working on the stale copy of F on the tablet, keep a separate transaction log L_2 of the work in the tablet, and then later on, when both machines happen to be up at the same time, have a way to reconcile L and L_2. In the best case, the copies can be reconciled automatically (e.g., the user is working on two different parts of the photo that can be just merged). In the worst case, manual conflict resolution is required. Option 2 is in fact the only option if there is truly no other way the devices can communicate with one another (e.g., if the user is on a plane with the tablet and with no network connectivity). However, it seems wasteful human effort that the user has to resort to this option even when the network bandwidth in many places (e.g., within the home, or work) would be perfectly adequate for automatic peer-to-peer sync, if only the devices were on.

Example 2: Device transparency: Several systems advocate *device transparency*, where the namespace reflects ones' files and data, not the device where they reside. Eyo, for example, allows a user to list from *any* device the metadata (e.g., name) of all files, residing in *all* subscribed devices [26]. We like the idea of the metadata being always available, but want to help further by satisfying the user's data needs as well. Imagine a user U having the names of all her documents, photos and videos, displayed on her tablet. When U meets a friend in the mall, she wishes to show her a short video from a birthday party. The video happens to physically reside on her home computer (although the metadata is on the tablet). There is reasonable 3G bandwidth to stream the video, but the home computer is dormant. The user knows the video exists, but cannot access it.

Example 3: Cloud storage: Having sufficient storage space to store *all* user data in the cloud with fast network connectivity to access it seems technically likely in the next few years (perhaps sooner in Silicon Valley). However, any consideration of data placement must include

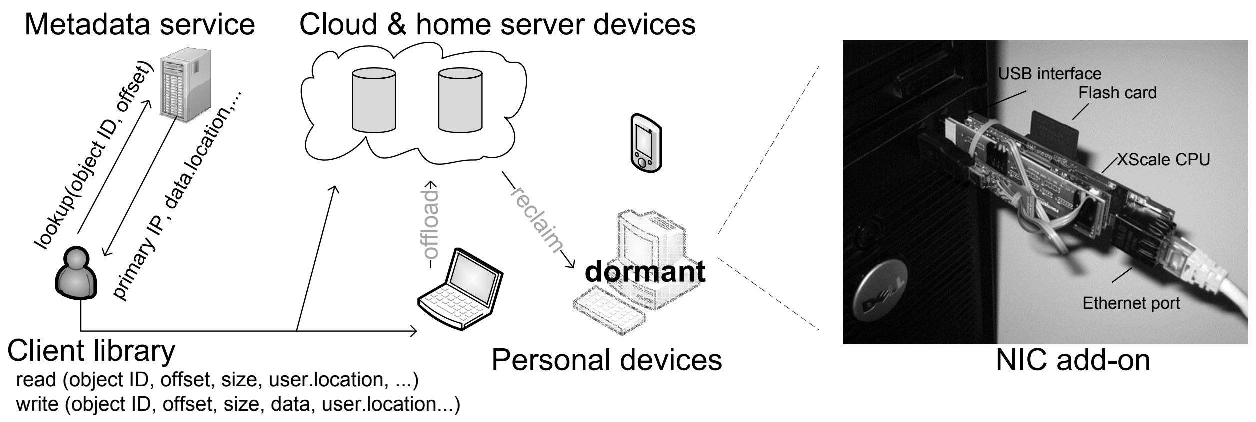
Metadata service
Cloud & home server devices
lookup(object ID, offset)
primary IP, data.location,...
offload
reclaim
dormant
Client library
read (object ID, offset, size, user.location, ...)
write (object ID, offset, size, data, user.location...)
Personal devices
USB interface
Flash card
XScale CPU
Ethernet port
NIC add-on

Figure 1: Storage system architecture, basic interfaces and Somniloquy hardware in action.

human factors as well as technology and cost trends. Human factors include, among others, trust in the cloud and desire to possess, know and control where one's data is located. Section 4 describes qualitative user studies we did in the context of this paper. From those studies, we believe that devices will continue to be places where users store some of their data. As such, we fully embrace the cloud as another place to store data, and we let users ultimately decide how to use that place. We do not second-guess them or force them to automatically place everything on the cloud. Internally, the system makes good use of available cloud space (e.g., for storing metadata — Section 3.2, and versioned logs — Section 3.3).

On the technical front, our system helps users who might have slow network connections to the cloud. Imagine a scenario in which a user decides to store a substantial amount of his data on the cloud. A user editing an article and compiling code while traveling benefits from the device's cache to batch writes before sending them to the cloud. When the user returns home and wants to continue working on the data from his home PC, he finds the PC's state is stale and incurs large performance penalties until the state is refreshed. A good cache placement policy would automatically hoard the user's working set to the home cache before the user would need to use it. Such a policy is hampered, however, because the home PC is likely dormant before the user arrives.

Intuition on how this paper helps: This paper is about enabling a satisfying execution of a user's data placement, caching and consistency policies given the likelihood that devices they rely on are dormant. One way our system will help the situation in Example 1 is by allowing peer-to-peer sync policies to work by turning devices on and off rapidly and automatically. If peer-to-peer sync would not be advisable (e.g., because of battery considerations), the system temporarily offloads the transaction log L onto the cloud. In Example 2, the system will continue to present a device-transparent view of metadata, and will rapidly turn on the home computer to get the data to the user. In Example 3, either peer-to-peer or cloud-to-device cache syncing will be enabled by turning the devices whose caches need refreshing on.

3 Design

Figure 1 shows several building blocks of the storage system. First, storage-capable devices strive to always maintain a low-power communication channel through a new low-power network card. Second, a metadata service maintains a unified namespace, encompassing any available storage space on devices, cloud and any home servers. Third, an I/O director, in cooperation with the metadata service and the new communication channel, manages the I/O flow through the system.

3.1 Maintaining network awareness

Data placement and consistency protocols are helped if devices maintain an always-on communication channel, even when dormant or off. Of course, such a channel should consume minimal power. We chose to use a new network interface card, called Somniloquy, that is designed to support operation of network-facing services while a device is dormant. Figure 1 shows it operating with one of our desktops. Somniloquy was first described by Agrawal et al. [2] in the context of reducing PC energy usage. The hardware is not a contribution of this paper. This paper reports on Somniloquy's role and integration into a distributed personal storage system.

Somniloquy consumes between one and two orders of magnitude less power than a PC in idle state. Somniloquy exports a 5 Mbps Ethernet or Wireless interface (Figure 1 shows a prototype with the Ethernet interface) and a few GB of flash storage. Somniloquy runs an embedded distribution of Linux on a low power 400 MHz XScale processor. The embedded OS supports a full TCP/IP stack, as well as DHCP and serial port communication. Power consumption ranges between 290 mW for an idle wireless interface, 1073 mW for the idle Ethernet interface, and 1675 mW when writing to flash [2].

Somniloquy allows a dormant device to remain responsive to the network. The NIC can continue to communicate using the same IP address as the dormant device. Somniloquy is more appropriate than Wake-on-LAN (WoL) [11] for mobile storage devices, because it operates through firewalls and NAT boxes, and it handles mobility across subnets. The on-board processor maintains contact with a DNS server to preserve the hostname-to-IP address mapping, performs basic networking tasks, and does I/O to its local flash card.

Does the new NIC make the overall system less secure? Our experience is incomplete. Logically, the system is running the same storage service as before. However, because parts of that service now run on the NIC's processor, the attack surface on the system as a whole has increased. Also, while modern processors have additional security features such as execute-disable bits to prevent buffer overflows, our low power processor does not support these features yet. Denial-of-service attacks might result in drained batteries. To partially mitigate these problems we force the NIC to only listen on one port (5124) that belongs to the storage service. Further, we require the main device and the NIC's processor to be on the same administrative domain.

Somniloquy is the hardware part of the solution, but it is insufficient without the storage and file system software. Here we give intuition on how the I/O director (Section 3.3) will use Somniloquy for two common operations: reads and writes. A read to a file on a Somniloquy-enabled storage device incurs a worst-case latency when the request arrives just as the device is going into standby. Somniloquy will wake up the device and the latency is at least *standby* + *resume* time. Table 1 shows some measurements to understand this worst-case penalty. Future devices are likely to have faster standby and resume times. Writes do not have a similar latency penalty. The I/O director can temporarily offload data to Somniloquy's flash card, or nearby storage-capable resources (such as the cloud) if these are available.

Summary, limitations and alternatives: We design to allow devices to maintain network awareness even when dormant. Our specific way of enabling the goal is to introduce new NIC hardware to each device. Agrawal et al. [2] describes why Somniloquy is more appropriate than several other hardware-based alternatives (e.g., Turducken [25]) and we do not list those alternatives further here. An assumption we make is that it is cost effective to augment devices with a smarter network interface card. Further, we assume the NIC would not drastically change the failure characteristics of the device. These assumptions might turn out to be a limitation of our approach, depending on the economics of producing a device and its failure characteristics. Another limitation is a lack of evaluation of Somniloquy with tablets or smartphones. Currently the driver works for Windows Vista/7 only, which limits the experiments in Section 5 to laptops and desktops. Currently, the NIC can only wake up devices that are placed into standby, and are not fully off.

Device	Standby(s)	Resume(s)
Lenovo x61 (Win7)	3.8	2.6
Dell T3500 (Win7)	8.7	7.2
HP Pavillon (XP)	4.9	10.25
Macbook Pro (OSX 10.6.8)	1	2
Ubuntu 11.10	11	4.5

Table 1: Example suspend and resume times for commodity devices. The device is first rebooted to clear previous state then it is put into standby followed by a resume. Section 5.2 shows more realistic end-to-end measurements using the Dell T3500 device.

A software-based alternative would be to maintain device network awareness by encapsulating a device in a virtual machine abstraction and then making sure the virtual machine (VM) is always accessible. SleepServer, for example, migrates a device's VMs to an always-on server before the physical device goes dormant [3]. This alternative might be more appropriate in enterprise environments where VMs are used and dedicated always-on servers are available, rather than for personal devices.

3.2 Metadata service

The metadata service maintains a mapping among an object/file ID, the devices that object is stored onto, and the replication policy used. The MDS uses a flat object-based API by default, where each object ID is an opaque 128-bit string. The metadata service (MDS) is a logical component, and it can reside on any device or server. The metadata service might be replicated for availability. Consensus among replicated services could be maintained through the Paxos protocol [22]. Furthermore, the data belonging to the service might be replicated for reliability, or cached on devices for performance. Data consistency needs to be maintained across the replicas.

The low-power communication channel in Section 3.1 helps with MDS availability and reliability in the following way. If the service is replicated among devices for availability, Somniloquy wakes up dormant devices that need to participate in the consensus protocol. If the data belonging to the MDS is replicated, the I/O director strives to maintain strong consistency through a range of techniques described in Section 3.3. A reasonable default for home users is to have a single instance of the metadata service run on a cloud server with content replication factor of 1, i.e., instead of being replicated, the metadata content is cached on all devices (this is what our file system implementation in Section 4 does). The metadata content can be cached on all devices since its size is usually small (Section 5.5).

The client library caches a file's metadata when a file is created and pulls metadata updates from the metadata service when accesses to a file fail. The latter could happen either because the file has moved or it has been deleted, the access control policy denies access, or the device has failed. A client's library synchronously updates the MDS when metadata changes. Those updates could be subsequently pushed by the metadata service to other devices caching the metadata (the push could be lazy, e.g., daily, or could happen as soon as the change occurs). For the common case when a device is dormant, Somniloquy could wake up the device (or absorb the writes in its flash card temporarily) to update its cache. A client might choose to pull the latest metadata explicitly (e.g., through a Refresh button), rather than using the push model. While the design supports both models, we believe a hybrid pull and lazy push model is a reasonable default for home users.

Our design requires storage devices to be explicitly registered with the MDS. If a device is removed from the system, either because it has permanently failed or because a newer device has been bought that replaces it, a user needs to explicitly de-register the old device and register the new device with the MDS. The metadata service initiates daily heartbeats to user devices to detect permanent failures and to lazily refresh a device's metadata cache. A heartbeat wakes up a dormant device. A device is automatically rebuilt after the user triggers the rebuild process.

Summary, limitations and alternatives: The novel aspect of our metadata service is that the execution of both metadata service consensus (for availability) and metadata replication consistency protocols (for reliability and performance through caching) is helped by the ability to turn participating devices on and off transparently. The design allows for several consensus and consistency options. However, by default the MDS resides on the cloud and its content is cached on all devices. The implicit assumption for this default is that the user will have at least ($>$56 Kbps) broadband connectivity at home or work and some weak 3G connectivity when mobile. Further, we assumed a few hundreds of MB of storage space at a cloud provider. We believe this is a weak assumption, but, even in the absence of cloud space, the metadata service and data could still reside on any device that incorporates Somniloquy.

3.3 I/O director

The I/O director is the third building block of our design. Its goal is to be versatile, allowing for a range of data placement and consistency policies. Uniquely to our system, the I/O director has new options for data movement. It can choose either to wake up a device to make reads or writes, or to temporarily use the flash storage provided by Somniloquy; it can also opportunistically use other storage resources to mask performance latencies and maintain the always-on communication channel.

The operations of the I/O director are best understood through Figure 2, which shows a client, a metadata service (MDS) and two devices D_1 and D_2. The data is replicated on both devices with a particular primary-based concurrency control mechanism to serialize concurrent requests. In this example, each replicated file has one replica that is assigned the primary role. Figure 2 shows some common paths for read and write requests. Reads, in the default case, are serviced by the primary for an object, as seen in Figure 2(a). When all devices are dormant and a read or write request arrives, Somniloquy resumes the device and hands it the request as shown in Figure 2(b) for reads and Figure 2(d) for writes, respectively. Writes are sent to the primary, which serializes them to the other replicas of the object as in Figure 2(d).

When objects are replicated and a device goes into a controlled standby, the metadata service receives an RPC indicating that, as seen in Figure 2(c). This is an optimization to give the MDS the chance to proactively assign the primary role away from that device to devices that are on. As might be expected, transferring the primary role does not involve data movement, just network RPCs to inform devices of the new assignment. A client's metadata cache might be stale with the old primary information, so a read will initially go to the dormant device. However, the device is not turned on, since the primary does not reside there. Instead, the client times out, which triggers an MDS lookup and cache refresh. The read then proceeds to the device with the primary, which happens to be on in this example.

The I/O director implements I/O offloading [15, 29] to mask large write latencies and to implement the logging subsystem. The logging subsystem gives the abstraction of a single virtual log to the whole distributed storage system. The actual log might reside on any storage device. Its size is limited by the size of cloud space, plus NIC flash space, plus all free hard drive space across all devices. Figure 2(e) shows offloading to the log (Section 5 evaluates the case when the log physically resides on a nearby device). Remember that if parts of the log are on the dormant device's hard drive, that device can be woken up as needed to access the log. Data is eventually reclaimed at the expected device at appropriate times,

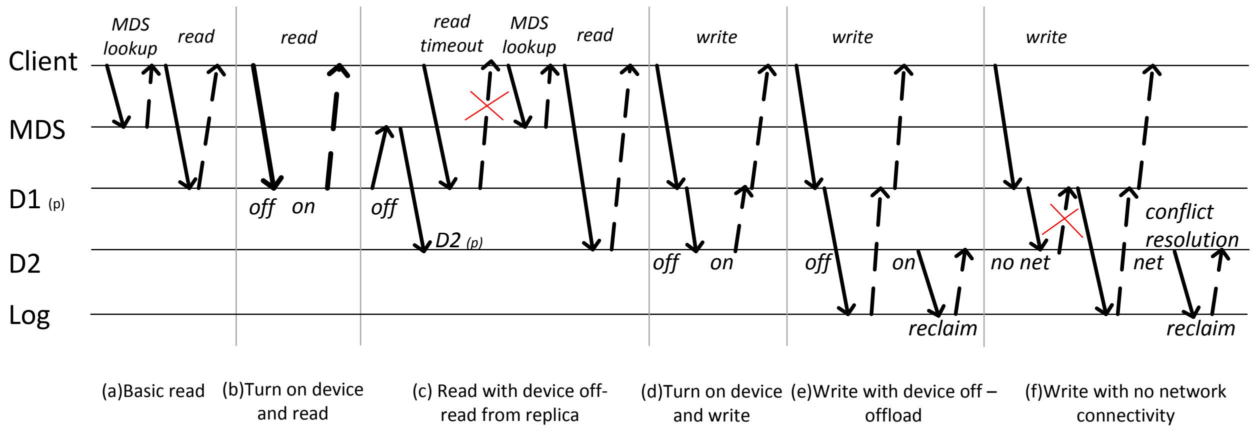
MDS lookup
read
read
read timeout
MDS lookup
read
write
write
write
Client
MDS
D1 (p)
D2
Log
off
on
off
D2 (p)
off
on
off
on
reclaim
no net
conflict resolution
net
reclaim
(a)Basic read
(b)Turn on device and read
(c) Read with device off-read from replica
(d)Turn on device and write
(e)Write with device off – offload
(f)Write with no network connectivity

Figure 2: Read and write protocols and common cases. *D* stands for device and *p* indicates that the primary for the file being accessed is on that device. "Off" and "on" indicate whether the device is dormant or not.

e.g., when the device is not in use.

The system is optimized for the common case when there is some network connectivity among devices and the cloud. If that is not the case, e.g., when the user is on a plane without network access, the system will temporarily offload all user writes to the log, and the log will have to physically reside with the user's device locally. When the user gains network connectivity, all participating devices will have to eventually reclaim data from the log and do standard conflict resolution (e.g., as in Bayou [28]), as illustrated in Figure 2(f). Our work does not add anything novel to this scenario's logic, but our implementation makes use of the existing logging infrastructure to keep track of write versions.

A user can move the file to a new device, and can change its replication policy any time. When any of these options happen, our system allows continuous access to the file. Any new writes to the file are offloaded to the versioned log. The I/O director logic maintains the necessary bookkeeping to identify the location of the latest version of a file. The location could be the old location, or the log, depending on whether the file has seen any new writes while being transferred or not. Once the file has moved to the new location, reclaim is triggered to copy any bytes that might have changed.

Summary, limitations and alternatives: The novel aspect of the I/O director is that it has new options for data movement. It can also choose to turn on a dormant device. The I/O director is optimized for an increasingly-common case of at least basic network connectivity among storage devices. It reverts to well-known conflict resolution techniques otherwise.

We currently use I/O offloading techniques [15, 29] to augment the base file system (which is not versioned) with a versioned file system partition. Ursa Minor's techniques for data placement versatility [1] are a good alternative in case the underlying file system is already versioned. For example, Ursa Minor uses backpointers when changing data replication while maintaining data availability. Also, advanced data encoding policies (e.g., the use of erasure codes), and other concurrency control methods (e.g., based on quorums) could equally benefit from our always-on communication channel.

3.4 Interaction with energy policies

As remarked above, Somniloquy consumes more than an order of magnitude less energy than an idle device while maintaining network awareness. The default interaction with energy policies is simple. A read overrides the energy policy and wakes up the device. Writes are fully buffered in the NIC's card or cloud before waking up the device. These policies are similar to the ones offered by BlueFS [16], in that they actively engineer and divert the traffic to the right device, but we have more resources available, in the form of the NIC's flash card or cloud.

Because the NIC runs a capable operating system, more complex energy policies can be encoded as part of the NIC processing. For example, BlueFS reduces energy usage by reading data from the device that will use the least amount of energy. That policy could be slightly modified to take into account the device turn on time, if the device is dormant. Furthermore, the storage system could determine whether to wake up a device or not as a function of whether the device is plugged in or running on batteries. Also, a more advanced standby strategy might predict future access patterns and prevent the computer from going into standby. Currently, our devices use simple idle time-based policies, like the ones implemented on Windows.

4 ZZFS: a file system artifact

Perhaps surprisingly, having a few extra always-on resources allows for interesting data placement policies that were not possible before. We explore these through building a file system called ZZFS. We chose to implement a unique set of data placement and consistency policies that cater mostly to spontaneous data accesses.

4.1 Design rationale

The design rationale for ZZFS is indirectly influenced by data from qualitative user research, comments on mailing lists of popular sync tools like Live Mesh [14] and Dropbox [4], and our desire to explore new policies. ZZFS's policies are different from those of say, Cimbiosys [21] or Perspective [23], but not necessarily "better" or appropriate in all cases.

Data from sync programs: To understand how users perceive consistency and conflict problems and how they rate them in fix-priority when compared to performance problems we collected and analyzed user feedback for Live Mesh [14] and Dropbox [4], two popular rsync tools. They serve as a rather coarse proxy for understanding consistency in the absence of a distributed file system. Feedback from the sync programs is heavily biased toward early adopters and technology experts, of course, but it is nevertheless helpful if only because of its volume (thousands of messages on public forum boards). Example 1 in Section 2 was influenced by this data.

Qualitative studies: Our first qualitative study helped us understand how people understand, organize, store, and access their content across different devices. The users for the qualitative studies were picked at random by a third-party company that specializes in user studies. We performed "guerrilla" (street) interviews with six people. We visited two family homes and we then invited two different families to a conference room (provided by the third-party company so that our identities would remain unknown to avoid perception bias) to further discuss concepts through storyboards. The raw data we collected is available upon request, but we have not put it in paper form yet. In parallel, we conducted a second, larger-scale study on issues around data possession [17].

How the data influenced us: This research influenced us to try harder to cater to the character of data access and device management displayed by ordinary (i.e., non technical) users. We interpret the data as suggesting that syncing and replication policies are compromised by the ways users store data, their *ad hoc* access of networks, and the priority given to social and economic matters of data management.

By default, ZZFS caters to spontaneous users with no data placement policies specified at all by default. No user effort is required to pre-organize data on devices (by hoarding, syncing, etc.) Data by default remains on the device where the user chose to first create it, with a replication factor of 1. Users showed a greater concern for and doubts about transferring data between devices than device failure. This could be interpreted as similar to Marshall's observation that only 5% of data loss is due to a device failure [12].

Whenever a file needs to be accessed, the device it is on is asked to provide access to that file. If the device is dormant, the device is woken up through the I/O director and the network-aware part of the device. For more advanced users who worry more about device failure and thus specify a higher replication factor for files, ZZFS strives to reduce the time it takes to reach consistency among replicas by data offloading and by waking up devices as described in Section 3.3.

We found that users made deliberate and intelligent decisions about wanting to silo their data on different devices and the cloud. From both user studies, we believe that devices will continue to be places where users store their data. Any consideration of data placement must consider human factors as well as technology and cost trends. Human factors include, among others, trust in the cloud and desire to possess, know and control where one's data is located. Furthermore, different devices have unique affordances [6] and properties (e.g., screen size, capacity, weight, security, price, performance, etc.). Users seem capable of understanding those affordances, and ZZFS does not second guess. Data movement is incurred only when a user explicitly chooses to do so.

4.2 Implementation details and status

We have implemented most of the design space described in Section 3. ZZFS is a distributed file system that results from picking a set of policies. It is implemented at user-level in *C*. ZZFS supports devices whose local file system can be NTFS or FAT. ZZFS has implemented per-object replication and allows for in-place overwrites of arbitrary byte ranges within an object. Concurrent accesses to a file are serialized through a primary. ZZFS's namespace is flat and it does not have folders, however it maintains collections of files through a *relate()* call.

The current implementation addresses a limited set of security concerns. Data and network RPCs can be encrypted (but are not by default). Each object has an access control list that specifies which user can access that object and from what device. We are actively doing research in what security means for home users [13].

In addition to simple benchmarks that directly access ZZFS through a client library, we run unmodified, legacy applications (e.g., MS Office, iTunes, Notepad, etc.) for demoing and real usage. We do so by mounting ZZFS as a block-device through the use of a WebDav service [31]. This technique required us to detour the WebDav service to use our APIs [9]. WebDav file semantics are different from NTFS semantics and often lead to performance inefficiencies (e.g., any time a change is made to a file, WebDav forces the whole file to be sent through the network). The following calls are detoured to use ZZFS's calls: *CreateFile()*, *FindFirstFile()*, *FindNextFile()*, *ReadFile()*, *WriteFile()*, *GetFileAttributes()* and *DeleteFile()*. The interface currently is Windows Explorer. A more appropriate interface for a distributed file system is work-in-progress.

ZZFS is robust. We are using it daily as a secondary partition to store non-critical files. When it crashes, it usually does so because of the NIC's device driver. The driver issues will be resolved over time and were not our primary focus for this paper. However, we are working toward having ZZFS as a primary partition for all files.

5 Evaluation

First, we measure how ZZFS performs and locate its bottlenecks. Second, through a series of real scenarios, we measure latencies and penalties associated with the always-on communication channel. This is an evaluation of the underlying storage system and also of ZZFS's policies. Third, we provide analytical bounds for performance for a range of workload and device characteristics. Fourth, we examine metadata scalability.

5.1 Exposing throughput bottlenecks

This section focuses on **throughput**. The other sections will focus on latency. We compare our system against local file access through the NTFS file system. This is the only time we will use a set of homogeneous devices (obviously not realistic for personal devices), because it is simpler for revealing certain types of bottlenecks. The devices are three HP servers, each with a dual-core Intel Xeon 3 GHz processor and 1 GB of RAM. The disk in each device is a 15 KRPM Seagate Cheetah SCSI disk. The devices have a 1 Gbps NIC. All reads and writes are unbuffered, i.e., we do not make use of the RAM.

First, we measure peak bandwidth and IOPS (I/Os per second) from a single device ("Read.1" and "Write.1" in Figure 3). Bandwidth is measured in MB/s using 64 KB sequential reads and writes to a preallocated 2 GB file. IOPS are measured by sending 10,000 random-access 4 KB IOs to the device with 64 requests outstanding at a time. Figure 3 shows the average from 5 results (the variance is negligible). Average local streaming NTFS performance (not shown in graph) is 85 MB/s for reads and writes and 390 IOPS for reads and 270 IOPS for writes; hence, ZZFS adds less than 8% overhead.

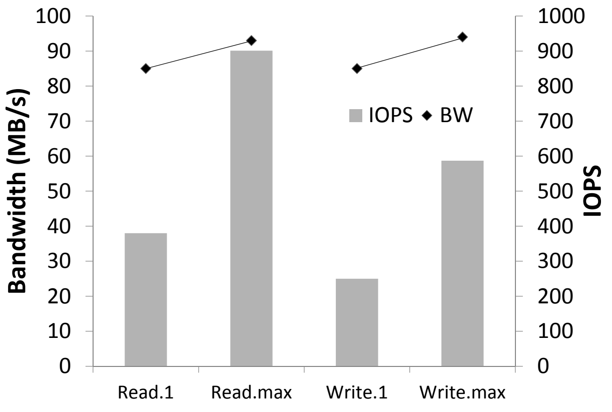

Figure 3: Baseline bandwidth and IOPS.

Second, we measure maximum bandwidth and IOPS from all three devices to understand performance scalability ("Read.max" and "Write.max" in Figure 3). Three clients pick one random 2 GB file to read or write to, out of a total of 10 available files. Each file is replicated 3-way. If all clients pick the same file, accesses still go to disk since buffering is disabled. Figure 3 shows the results. As expected from 3-way replication, the saturated write bandwidth is similar to the bandwidth from a single device. Saturated read bandwidth is about a third of the ideal because requests from all three clients interfere with one another. This problem exists in many storage systems because of a lack of performance isolation [30]. Saturated IOPS from all devices is close to the ideal of 3x the IOPS from a single device.

Overall, these results show that our system performs reasonably well with respect to throughput. Optimizations are still required, however, especially with respect to reducing CPU utilization. CPU utilization in the saturated cases was close to 100%, mostly due to unnecessary memory copies.

5.2 I/O director

This section focuses on read and write **latencies** resulting from the always-on channel. We have real measurements from a home wireless network. We start by illustrating the performance asymmetry between reads and writes. The first workload is an I/O trace replay mimicking a user listening to music. We use trace replay to just focus on I/O latencies and skip the time when the user listens to music and no I/O activity is incurred. Half of the music is on his laptop, half on the desktop and the setting is in "shuffle mode" (i.e., uniform distribution of accesses to files). The music files are not replicated. The desktop (Dell T3500 in Table 1) is on a 100 Mbps LAN and the laptop (Lenovo x61) is on a 56 Mbps wireless LAN. The Somniloquy NIC is attached to the desktop. The MDS

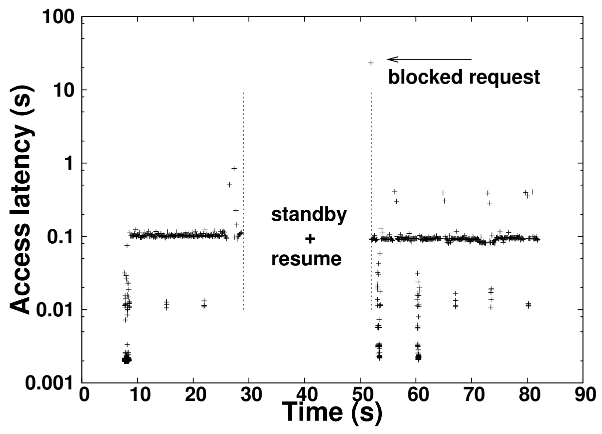

Figure 4: A scatter plot over time for a client's requests. Reads latencies are [mean=0.09 s, 99th=0.36 s, worst=23.3 s]. Write latencies are [mean=0.014 s, 99th=0.022 s, worst=0.058 s]. There are several performance "bands" for local reads (0.001-0.01 *s*), remote writes (0.05-0.1 *s*) and remote reads (0.05-1 *s*).

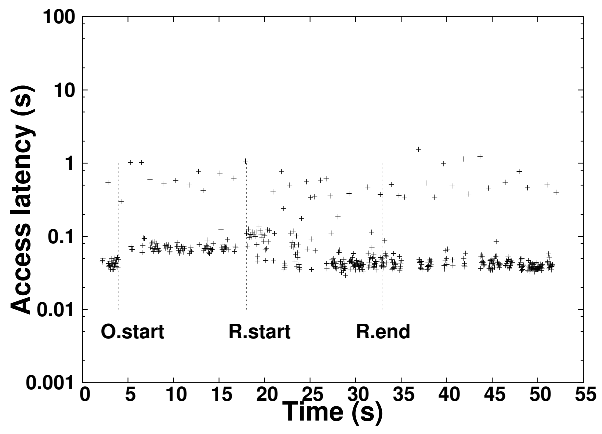

Figure 5: A scatter plot over time for a client's write requests. *O.start* annotates the time the second device enters standby, and thus offloading begins to a third device. *R.start* annotates the time when the second device resumes and reclaim starts (offloading thus ends). *R.end* annotates the time when all data has been reclaimed. Write latencies are [mean=0.1 s, 99th=1 s, worst=1.5 s].

resides on the desktop, but all metadata is fully cached on the laptop as well. The music program issues 64 KB reads to completely read a music file, then, after the user has finished listening, a database is updated with a small write of 4 KB containing ratings and play count updates. The database resides on the desktop and is not replicated. Hence, although this is a common workload, it is quite complex and has both reads and writes. The user simply wants to listen to music without worrying where the music files and database are located.

Figure 4 shows a scatter plot (and latency distribution in the caption) of the worst-case scenario when request to read a music file comes just as the desktop is starting to go into standby. Somniloquy intercepts the read request and signals the computer to wake up. The time it takes the computer to accept the request is 23.3 s (*standby time + resume time*) and is illustrated in the scatter plot in the figure. In practice, prefetching the next song would be sufficient not to notice any blocking; however, when prefetching is not possible, this serves as a worst-case illustration. We note that the desktop is rather old, and if using a newer device (e.g., the Macbook Pro in Table 1) the worst case latency would be around 4 seconds.

Figure 5 illustrates that writes do not suffer from this worst case scenario. The workload in this scenario is a trace replay of I/O activity mimicking a user sending 64 KB writes to a document from the laptop. The user uses 2-way replication for those files, with the second replica kept on the desktop. Both laptop and desktop are on the wired LAN. Similar to the previous case, the desktop has gone abruptly into standby. However, there is a second laptop nearby that is on, and the I/O director temporarily offloads the writes onto that laptop (other options for the offload location are Somniloquy's flash card or the cloud). This way, 2-way, synchronous replication is always maintained. When the desktop comes out of standby, the data on the third laptop is reclaimed. Reclaim does not lead to a noticeable latency increase. The figure shows a slight increase in latency during data offload since the second laptop is on the wireless LAN. A handful of requests experience high latencies throughout the experiment. We believe these are due to the performance of the wireless router. Note that writes in this experiment are slower than in Figure 4 because of larger write request sizes (64 KB vs. 4 KB) and 2-way replication vs. no replication.

We compare our system against simple *ping* and average disk latencies, i.e., we set a relatively high bar to compare against. We measured a minimum of 0.06 s ping latency for 64 KB[1], 0.005 s for 4 KB sizes and the disk's average latency is 0.015 s (these are slow SATA disks, not the fast SCSI disks used in the previous section). Hence, an end-to-end read request (and ack) should take on average 0.075 s and an end-to-end write request (and ack) should take on average 0.02 s[2]. Looking at the performance "bands" in Figure 4, we see that local read latency and remote write latency is very good, while remote read latency is 33% slower than ideal. We have

[1]Exact size is 65500 B, the maximum ping size.

[2]Although read requests are sequential, the disk head incurs at least a full disk rotation before receiving the next request, since the requests are sent one at a time. Also, experienced disk average latencies are sometimes better than the above theoretical value because our disk is not full and the files are on its outer tracks.

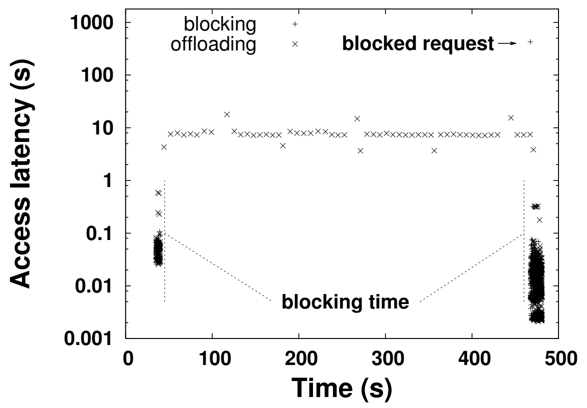

Figure 6: A scatter plot over time showing the effects of moving a file on concurrent operations on that file. Without offloading, the concurrent workload blocks; with offloading, the concurrent workload makes progress. When offloading, read latencies are [mean=0.7 s, 99th=8.6 s, worst=17 s]. Write latencies are [mean=0.5 s, 99th=7.3 s, worst=14 s].

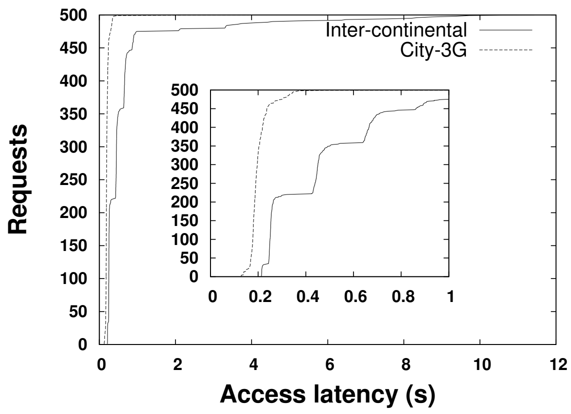

Figure 7: A CDF plot for a client's read and write requests over a 3G city-wide network and intercontinental network. For the 3G network, read latencies are [mean=0.21 s, 99th=0.35 s, worst=3.39 s]. Write latencies are [mean=0.17 s, 99th=0.3 s, worst=0.3 s]. For the intercontinental network read latencies are [mean=0.7 s, 99th=8.2 s, worst=11 s]. Write latencies are [mean=0.2 s, 99th=0.4 s, worst=0.4 s].

started collecting detailed performance profiles, but we note that the delay is unnoticeable to the applications.

File move: The next experiment demonstrates how moving an object affects performance of concurrent operations on that object. As discussed in Section 3.3, instead of locking the file for the duration of the move, the I/O director offloads any new writes to the file while the copy is in progress. In this experiment, we move a 1 GB file from one device to another while simultaneously running a series of 64 KB read and write (with a 1:1 read:write ratio) operations on that object. Figure 6 shows that, with offloading turned off, the read and write operations must block until the data move is complete; with offloading turned on and another laptop temporarily absorbing new writes, these operations make progress. The devices are limited by the 56 Mbps wireless LAN, and the network is saturated during the file move, hence access performance during that time is slow (around 10 s). We believe this is better than blocking for more than 400 s (the latency of "blocked request" in the figure). Note that after the move completes, performance improves because the client is co-located with the device the file is moved onto.

5.3 ZZFS's placement policy

Next, we measure ZZFS's performance in a 3G city-wide network and an intercontinental network. We look at the performance resulting from the simple policy of leaving data on the device it was first created. We illustrate the performance of our system when a user on the move is accessing music files stored on the home desktop. Unlike the music scenario above, the client has no music files or metadata cached on the laptop and always reads and writes to the home desktop. Access sizes are the same as before (64 KB reads and 4 KB writes).

First, when the user is on a city-wide 3G network, she is connected to the Internet through a ZTE MF112 mobile broadband device connected to her laptop. Figure 7 shows the latency results. The first request incurs a first-time setup cost from the 3G provider, which is also the worst-case latency (we do not know what the provider is doing; subsequent runs do not incur this penalty, but we show the worst case). We measured a minimum of 0.23 s ping latency for 64 KB sizes, 0.13 s for 4 KB sizes in this environment, and ZZFS's overhead is comparable. The latency is good-enough for listening to music.

Second, when the user is on the west coast of the US (Redmond, Washington) she is connected to the Internet through a 56 Mbps wireless LAN. The location of the music files is on a desktop in Cambridge, UK. Figure 7 shows the results. We measured a minimum of 0.25 s ping latency for 64 KB sizes, 0.19 s for 4 KB sizes in this environment. ZZFS's write overhead is comparable, but its average read latency is 60% higher than ping. We believe this is due to the unpredictable nature of the intercontinental network. Nevertheless, the user does not perceive any noticeable delay once the music starts.

A takeaway message from this section is that ZZFS's performance is good enough in all cases for the applications involved. Data is never cached in these experiments, so we expect even better performance in practice.

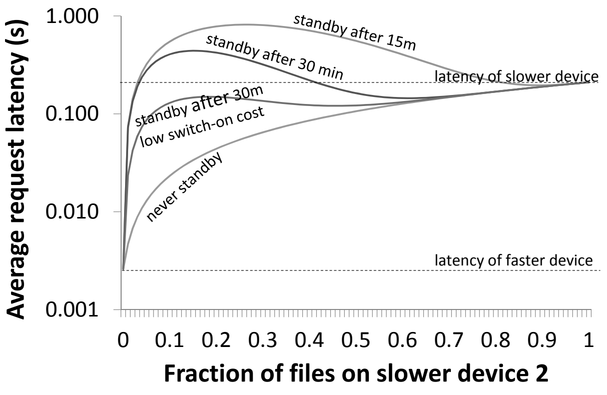

Figure 8: Latency tradeoffs for a client's read requests.

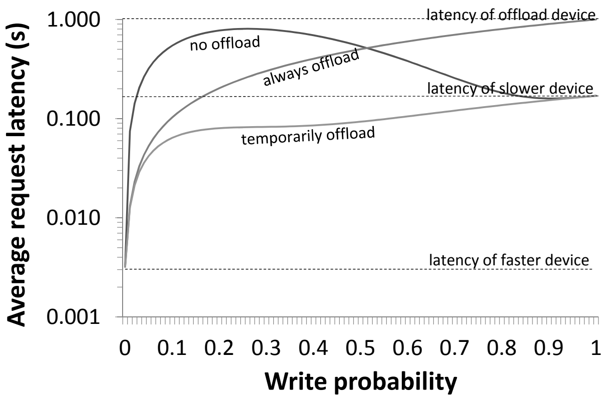

Figure 9: Latency tradeoffs for a client's requests when data is replicated on both devices.

5.4 Sensitivity to parameters

This section reexamines the above scenarios and others *analytically* while changing several tunable parameters.

In the next analysis, we revisit the music scenario. We still have two devices D_1 and D_2. First, we vary the amount of idle time I before D_1 enters standby (D_2 never enters standby since it is the device with the music player). Without loss of generality, we assume D_1's average access latency when D_1 is on, $L_1{}^{ON}$, is slower than D_2's average access latency L_2 (e.g., D_1 could be on the 3G network). $L_1{}^{STDBY}$ is the average access latency when D_1 is on standby. It is the time to resume the device plus $L_1{}^{ON}$.

Second, we vary the fraction of files p_1 that reside on the slower device ($p_2 = 1 - p_1$). For example, if D_1 enters into standby after $I = 15$ idle minutes and each song is on average $M = 5$ minutes in length, D_1 will enter standby if at least $\lfloor I/M \rfloor = 3$ consecutive songs are played from D_2 (with no loss of generality, we assume the writes go to a database also on D_2 this time, otherwise D_1 will never enter standby). Figure 8 shows the expected average latency given by:

$$\begin{aligned} E[L] = E[L|D_1 = ON]p\{D_1 = ON\}+ \\ E[L|D_1 = STDBY]p\{D_1 = STDBY\} \end{aligned} \quad (1)$$

The above equation further expands to $E[L] = (p_1 L_1{}^{ON} + p_2 L_2)p\{D_1 = ON\} + (p_1 L_1{}^{STDBY} + p_2 L_2)p\{D_1 = STDBY\}$. The analysis assumes a user is forever listening to songs, and this graph shows the long-running latency of accesses. All the lines assume the switch-on times of the Dell T3500, except for the low switch-on cost line that is the Mac.

We make several observations. In both extremes, where all files accessed are on D_2 or all files are on D_1, the latency is simply that of D_2 or D_1 respectively. If a device goes into standby, the worst latency tends to happen when the user accesses it infrequently, thus giving it time to standby and then resuming it.

The next analysis examines the impact of the read:write ratio of the workload. 2-way replication is used, and the same arguments are made about standby. The difference is that, in this case, D_1 enters standby if there are consecutive reads on D_2 (a write would wake up D_1 since it needs to be mirrored there.) Without loss of generality, we assume a read or write comes every 5 minutes and D_1 enters standby after $I = 15$ minutes. We illustrate the impact of turning on the device vs. always offloading (unrealistic in practice) vs. temporarily offloading while the device switches on. We assume without loss of generality that data is offloaded to a slow device, e.g., a data center.

Figure 9 shows the expected average latency $E[L]$ (a similar formula to the previous example is used, but the standby latency is the offload latency). We make several observations. For an all-read workload all files are read from D_2 (faster device). For an all-write workload the latency is determined by the slowest device. This slower device is either the offload device, if we always offload, or D_1. In all cases, offloading masks any switch-on costs.

5.5 Metadata

Table 2 shows the number of files for four families the authors of this paper are part of. This data is biased towards families with tech-savvy members. However, the point we make in this section is not that this data is representative of the population at large. We only confirm an observation made by Strauss et al. [26] that the amount of metadata involved is small in all cases and could easily reside in a data center today, and/or be fully cached on most consumer devices. We do this while showing that ZZFS's metadata structures are reasonably efficient.

We measured the amount of data with $R = 1$ and extrapolated for $R = 3$. The amount of metadata is calculated from ZZFS's metadata structures and is a function of the replication factor and number of files. It is interesting to observe that the second family has relatively fewer media files, and hence the average file size is much smaller than the other families. This translates to a higher relative metadata cost. Intuitively, the ratio of metadata to data decreases with larger file sizes.

6 Related work

Data placement on devices and servers: AFS [8] and Coda [10] pioneered the use of a single namespace to manage a set of servers. AFS requires that client be connected with AFS servers, while Coda allows disconnected operations. Clients cache files that have been hoarded. BlueFS [16] allows for disconnected operation, handles a variety of modern devices and optimizes data placement with regard to energy as well. EnsemBlue [19] improved on BlueFS by allowing for a peer-to-peer dissemination of updates, rather than relying on a central file server. In Perspective, Salmon et al. use the *view* abstraction to help users set policies, based on metadata tags, about which files should be stored on which devices [23]. Recent work on Anzere [22] and PodBase [20] emphasizes the richness of the data placement policy space for home users.

An implicit assumption of the above work is that home users know how to set up these policies. This assumption might have been borrowed from enterprise systems, where data placement decisions can be automated and are guided by clear utility functions [27]. Our low-power communication channel can help with the execution of the above policies and can be used by most of the above systems as an orthogonal layer. It ensures that devices are awoken appropriately when the storage protocols need them to. While our design is compatible with the above work, ZZFS's choice of specific policies for data placement is arguably simpler than in the above work. It stems from our belief that, for many users, it takes too much time and effort to be organized enough to specify placement and replication policies like in Perspective or Anzere. ZZFS shows that in many common cases, no user involvement is required at all.

Consistency: Cimbiosys [21] and Perspective [23] allow for eventual consistency. Cimbiosys permits content-based partial replication among devices and is designed to support collaboration (e.g., shared calendars). Bayou [28] allows for application-specific conflict resolution. Our work can help the user's perception of consistency and reduces the number of accidental conflicts. In a system with eventual consistency, the low-power communication channel can be seen as helping reduce the "eventual" time to reach consistency, by turning dormant devices on appropriately.

File system best practices: ZZFS builds on considerable work on best-practices in file system design. For example, our distributed storage system has a NASD-based architecture [7], where metadata accesses are decoupled from data accesses and file naming is decoupled from location. The system is device-transparent [26]. The I/O director maintains versioned histories of files that can later be merged and is based on I/O offloading [15, 29].

Family	R	#files	data(GB)	metadata(MB)-%
1	1	23291	582	11 (0.0019%)
	3	23291	1746	68 (0.0038%)
2	1	3177	2.44	1.6 (0.06%)
	3	3177	7.32	9.3 (0.12%)
3	1	31621	705	15 (0.002%)
	3	31621	2116	93 (0.004%)
4	1	124645	164	61 (0.036%)
	3	124645	492	365 (0.07%)

Table 2: In ZZFS, the size of metadata is *O(numfiles x numdevices)*. This table shows the **total** data and metadata size for existing files of some of the authors. Files included are "Documents," "Pictures," "Videos" and "Music." *R* is the replication factor.

User-centered design: We were inspired by a user-centered approach to system design. This was manifest not only in undertaking a small version of user research ourselves (Section 4), but by reference to the findings in the HCI literature in general. This literature still remains small on the topic dealt with here (e.g., see [20, 23] and also [5, 13, 17, 18]), but nevertheless helped provide some of the insights key to the technical work which is the main contribution of the paper.

7 Summary

Unpredictable networks and user behavior and non-uniform energy-saving policies are a fact of life. They act as barriers to the execution of well-intended personal storage system policies. This paper's contribution is to manage better these inherent uncertainties. We designed to enable a world in which devices can be rapidly turned on and off and are always network aware, even when off or dormant. The implications for the file system were illustrated through the implementation of ZZFS, a distributed device and cloud file system, designed for spontaneous and rather *ad hoc* file accesses.

8 Acknowledgments

We learned from the feedback of our shepherd Atul Adya, the anonymous reviewers, Tim Harris and Austin Donnelly, as well as several others at Microsoft.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, Dec. 2005.

[2] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: augmenting network interfaces to reduce PC energy usage. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 365–380, Boston, Massachusetts, 2009.

[3] Y. Agarwal, S. Savage, and R. Gupta. Sleepserver: a software-only approach for reducing the energy consumption of PCs within enterprise environments. In *Proceedings of the 2010 USENIX annual technical conference*, Boston, MA, 2010. USENIX Association.

[4] Dropbox. Dropbox. `https://www.dropbox.com`.

[5] W. K. Edwards, M. W. Newman, and E. S. Poole. The infrastructure problem in HCI. In *CHI '10: Proceedings of the International Conference on Human factors in Computing Systems*, Atlanta, GA, 2010.

[6] W. W. Gaver. Technology affordances. In *CHI '91: Proceedings of the International Conference on Human factors in Computing Systems*, New Orleans, Louisiana, United States, 1991.

[7] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 1998.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6:51–81, February 1988.

[9] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. USENIX Windows NT Symposium*, Seattle, WA, July 1999.

[10] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[11] Lieberman software. White paper: Wake on LAN technology. `http://www.liebsoft.com/pdfs/Wake_On_LAN.pdf`.

[12] C. Marshall. Personal archiving 2011 keynote: People are people and things change. `http://research.microsoft.com/en-us/people/cathymar/pda2011-for-web.pdf`.

[13] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter. Access control for home data sharing: Attitudes, needs and practices. In *CHI '10: Proceedings of the 28th International Conference on Human Factors in Computing Systems*, Atlanta, Georgia, USA, 2010.

[14] Microsoft. Windows Live Mesh. `http://explore.live.com/windows-live-mesh`.

[15] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, 2008.

[16] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the Blue File System. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 363–378, San Francisco, CA, 2004.

[17] W. Odom, A. Sellen, R. Harper, and E. Thereska. Lost in translation: Understanding the possession of digital things in the cloud. In *CHI '12: Proceedings of the International Conference on Human factors in Computing Systems*, Austin, TX, 2012.

[18] W. Odom, J. Zimmerman, and J. Forlizzi. Teenagers and their virtual possessions: Design opportunities and issues. In *CHI '11: Proceedings of the International Conference on Human factors in Computing Systems*, Vancouver, Canada, 2011.

[19] D. Peek and J. Flinn. EnsemBlue: integrating distributed storage and consumer electronics. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, Seattle, WA, 2006.

[20] A. Post, J. Navarro, P. Kuznetsov, and P. Druschel. Autonomous storage management for personal devices with PodBase. In *Proceedings of the 2011 USENIX annual technical conference*, Portland, OR, 2011. USENIX Association.

[21] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: a platform for content-based partial replication. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 261–276, Boston, Massachusetts, 2009.

[22] O. Riva, Q. Yin, D. Juric, E. Ucan, and T. Roscoe. Policy expressivity in the Anzere personal cloud. In *2nd ACM Symposium on Cloud Computing (SOCC)*, Cascais, Portugal, 2011.

[23] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *In Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, 2009.

[24] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: A distributed mobile storage system. In *In Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 239–252, San Francisco, CA, 2004.

[25] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: hierarchical power management for mobile devices. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 261–274, Seattle, Washington, 2005. ACM.

[26] J. Strauss, J. M. Paluska, C. Lesniewski-Laas, B. Ford, R. Morris, and F. Kaashoek. Eyo: device-transparent personal storage. In *Proceedings of the 2011 USENIX annual technical conference*, Portland, OR, 2011. USENIX Association.

[27] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, San Jose, California, 2008. USENIX Association.

[28] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, United States, 1995.

[29] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of Eurosys'11*, pages 169–182, Salzburg, Austria, 2011. ACM.

[30] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *In Proc. USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, 2007.

[31] Webdav.org. Webdav resources, 2010.