Accord: Middleware Support for Contextual, Ubiquitous Data Management on User Devices

Brian F. Cooper, Charles L. Isbell, Jeffrey S. Pierce, David L. Roberts, and Sooraj Bhat

College of Computing, Georgia Institute of Technology
{cooperb,isbell,jpierce,robertsd,sooraj}@cc.gatech.edu

Abstract. People increasingly use a diverse array of computational devices, including desktop PCs, one or more laptops, a cell phone, a PDA, tablet PCs, digital music players, automobile computers, and so on. We present *Accord*, a middleware system we have implemented to manage user data across all of these devices. Accord emulates an ideal abstraction we call a *user data-space*: a virtual space in which user files exist independent of any particular physical device. Users put files into the space with whatever device is convenient, and later access those files using any of their devices. This abstraction is difficult to implement, and requires Accord to predict when a file will be needed and on which device. We describe two mechanisms the middleware uses to support such predictions: an *object graph*, which records contextual and statistical information about file objects, and a *file transfer planner*, which uses predictions to determine how to efficiently move files between devices despite connectivity, bandwidth and storage constraints. Predictions can be constructed using simple usage statistics, or from more complex machine-learned models of user activities. We also present experimental results demonstrating the effectiveness of our system.

Keywords: mobile devices, disconnected operation, file replication

1 Introduction

People are faced with an increasing array of personal computing devices. Users might have devices ranging from multiple desktops and laptops to a PDA, a wireless email client, cellphone and tablet PC. Continued advances in technology will increase the power and variety of these devices; already, users are able to extend their personal network to include in-automobile computers, wearable computers and digital music players.

While the increasing ubiquity of devices enhances users' access to computation, it sharply decreases the ease of managing their data. In particular, users do not carry all of their devices with them, and therefore must frequently transfer files between devices so they have the information on the right device at the right time. People often find this process difficult to perform correctly. A user might create a document on his home desktop and then later need it on his work

laptop. Or he might edit one version of a document on his work desktop, make a few changes on his PDA, make a few more changes on his laptop, and then later have difficulty integrating all of the changes or even figuring out whether he edited the most recent version on each device. A user may maintain her contact information on her cellphone, and then accidentally leave it at home one day, making it impossible to find her doctor's phone number in order to change her appointment.

These examples illustrate two intertwined issues: the difficulty of predicting in advance which data will be needed on which device, and the challenge in synchronizing and managing many pieces of data across several heterogeneous devices. Existing tools are often insufficient: manual tools (such as shared directories, FTP or SCP) only work if the user remembers to use them, and use them correctly; automatic tools (such as "hotsync" software for PDAs) typically work for only a subset of the user's devices. Users need a comprehensive solution that works for all of their data and all of their devices.

We have developed a middleware system, called *Accord*, for automatically managing user data across a wide variety of heterogeneous devices. Accord emulates an abstraction we call a *virtual data-space*: a logical storage system that is independent of any physical device. Users store data in the data-space using one of their devices, and are then able to retrieve it using any of their devices. In this abstraction, devices are not storage containers but rather windows into the data-space, and users can use whichever of their devices is most convenient to access the data-space at any time. In a sense, our middleware brings disparate devices into accord with one another to serve as a single, harmonized data-space.

This ideal abstraction is difficult to achieve. In particular, storage must reside somewhere. However, many users do not have their own fileserver, and often none of their devices are appropriate to serve as a fileserver (because they are not always connected, not always on, or not capable enough). Therefore, we must synthesize a virtual data-space from multiple independent storage containers (the devices themselves). We face two main challenges in developing software to implement a user data-space:

- Each device has constraints on its storage resources. Some devices (such as desktops) may have a large amount of storage, while others (cell phones and PDAs) likely have very limited storage.
- Connectivity between devices is constrained and intermittent. A desktop at a user's home on a DSL line may have permanent connectivity to the user's work PC, but only limited bandwidth to the user's cellphone, and only intermittent connectivity to the user's wireless laptop or PDA.

Because of these limitations, simple approaches to building a virtual, ubiquitous user data-space will not work. For example, automatic synchronization that replicates all files to all devices is not feasible, given resource limitations, and fetching missing files on demand will only occasionally work given connectivity limitations. Moreover, our solution must deal with several other properties of user devices: devices can get turned on and off, computing power varies widely between devices, and devices may fail. In particular, some devices (such as USB

keys) have no processing capability at all; yet, we still want to use these "dumb" devices as part of the system because the user is likely to be carrying them much of the time. As such, the challenge is to develop a middleware system that provides a useful implementation of a virtual, ubiquitous user data-space despite the limitations of user devices.

Our approach is to build a system that gathers as much context and statistical information as possible about file usage, and then uses this information to make predictions about when and where a file will be needed. For example, if the system notices that a spreadsheet was edited on a user's home computer, and predicts that it will next be edited on the user's laptop, it can proactively move the spreadsheet to the laptop. In the optimal situation, if the user tries to access a file on a given device, he will only fail if it was impossible to proactively move that file to that device given resource constraints. Such predictive file movement requires several components: a distributed data structure for managing the context information, machine learning techniques to make the predictions, a file transfer component that can move files to devices based on predictions, and a user feedback module to correct or enhance the predictions.

Here we focus on the system-level components of Accord; specifically, the distributed contextual data structure and file transfer planning. Techniques for machine learning and user feedback interfaces are the subject of ongoing research in our group, and can now be supported by our implemented Accord middleware. For example, machine learning techniques can only be effectively tested when we have a large amount of actual data to learn from, and our implemented prototype can now be used to gather that data from real users (ourselves included). However, our experimental results demonstrate that even with very simple usage statistics (e.g., the number of reads and writes of each file on each device) Accord can significantly improve the synchronization of data across devices compared to traditional epidemic replication.

In this paper we describe the design and implementation of Accord. First, we discuss the requirements and architecture for Accord (Section 3). Then, we describe the middleware services that Accord provides, including an *object graph* for representing contextual information across user devices, and an *adaptive file transfer planner* for robustly moving files between devices despite connectivity, bandwidth and storage limitations (Section 4). Next, we present experimental results that demonstrate that using Accord imposes low overhead, while significantly improving the probability that the right files are available on the right devices (Section 5). We also survey related work (Section 2) and discuss our conclusions (Section 6).

2 Related work

The classic system for managing information on mobile devices is Coda [1]. Coda extends the Andrew network filesystem to provide support for disconnected operation. We aim to extend and improve upon the techniques in Coda in several ways. The key difference is that Accord is targeted at using machine learning

techniques to predict where files will be needed. Coda's hoard profiles determine where files will be replicated, and are either manually constructed or derived using heuristics. By using machine learning, we hope to provide effective replication despite wide variation in the ways in which a user utilizes different devices. Other differences with Coda include:

- Support for users who do not have a device that can act as a central file server (because none of their devices is permanently connected, always on or capable enough).
- Adaptivity to widely varying storage capacities and computational capabilities of different devices. We also support "dumb" devices (such as USB keys) that have no processing capability.
- Support for scenarios where disconnected operation is the common state. Coda assumes disconnection from the central server is temporary. When disconnection is the norm, multiple hops over several devices may be needed to accomplish file transfers.

The Odyssey system [2] extends Coda to provide application-aware optimizations (such as image downsampling). Our goal is to provide a transparent data management layer. Thus, while application-specific optimizations may be incorporated in the future, our goal is to provide high performance even without declared application properties (perhaps by learning such properties from context information).

There has been a large amount of other work on update consistency in disconnected or weakly connected networks. Examples include FICUS [3], Little Work [4], Bayou [5], Thor [6], and others. The goal of most of these systems is to manage updates, propagating them to a primary copy or to various clients. Our goal is somewhat different: to proactively place cached copies of files where a user will need them. Once the replicas are placed, standard techniques can be used or extended to manage consistency issues. Segank [7] combines update management with functionality to search a device network for replicas. Accord's object graph provides replica location information, which could potentially be augmented with Segank-style multicast of document requests. Segank also assumes that devices are always connected, while Accord deals with devices where disconnection is the frequent state.

Some systems dispense with the central file server altogether. For example, xFS is a serverless network filesystem [8]. xFS assumes a high speed network and is appropriate for a network of workstations. Different techniques must be developed for a serverless network filesystem for mobile and disconnected devices. Peer-to-peer overlays [9, 10] and filesystems [11, 12] provide support for locating objects, but do not provide inherent capabilities for proactively caching objects on devices to support mobile or disconnected operation.

A large amount of work has focused on building mobile and disconnected systems. An overview of early work is provided in [13]. More recent work has focused on user-centric routing of messages [14], connectivity management [15], session movement [16–18] and multihop routing in ad hoc networks [19, 20]. Much of this

work focuses on low level issues, such as routing packets or managing sessions. As such, it is complementary to our higher level focus on user-level data and its associated context. SyD [21] is a middleware system for application development on mobile devices. SyD focuses on collaborative groupware and provides support such as distributed transactions. Transactions are not our primary focus, although we could extend our system to provide transaction support. Satchel [22] utilizes a special device (a Nokia 9000 Communicator) as a central coordinator for document management. Accord does not require the user to own or carry a particular device, and continues to function even if any particular device fails.

Several investigators have examined techniques for using contextual information to support machine learning [23, 24] and enhance user interfaces [25, 26]. In our ongoing work we are extending the techniques in these systems to utilize the context information provided by Accord.

3 System design

3.1 The goal of a virtual data-space

The *virtual data-space* abstraction ideally provides the following guarantee: for a single user interacting across his own devices, our system will provide local access to any file that he wants to access on a particular device. Note that this does not require replicating all information on all of a user's devices, because the type of device will constrain the files a user wants to access. For example, a user is unlikely to read and modify a long document on his cell phone.

However, two properties of user devices make it difficult to implement a true virtual data space. First, connectivity between devices is variable, ranging from persistent DSL connections to sporadic WiFi and Bluetooth connections. Second, devices have widely varying storage and computational capabilities. In particular, not every device can store every file. Therefore, Accord can only really implement a *best-effort virtual data-space*, attempting to maximize the probability that a user can access any file locally on any device. Sometimes, Accord will be unable to move the data to the proper place. For example, there is very little Accord can do to ensure the right data is on a device that is never connected to any other device. We rely on the user interface to alert the user to such pathological cases, while designing Accord to provide the best service for more common cases.

We do not expect the user to have a very large number of devices. A survey of Georgia Tech users we conducted found that people had on average 4.7 devices. However, the study also showed that it was common for users to have heterogeneous devices, including one or more desktops, a laptop, a PDA and a cell phone.

3.2 Architecture

Our approach to implementing a virtual data-space is for the middleware to collect contextual information about which files are used, when, and where, and

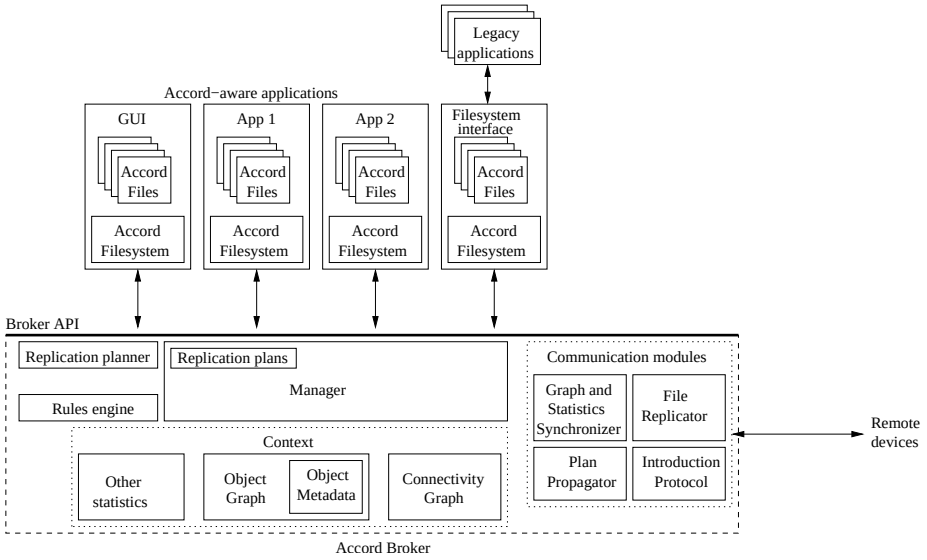


Fig. 1. Accord middleware architecture.

then use this information to proactively transfer files to devices before the user needs them. In particular, the contextual information allows us to infer user activity patterns, either at a very rough level (e.g. "file X is used frequently on device Y") or at a more fine-grained level (e.g. "with probability p, file X is used between 3-5 p.m. on device Y if it is a weekday"). Our current system uses simple usage statistics to make replication decisions. Now that our prototype is implemented, it can be used to support research in machine learning and HCI to infer more complex user patterns (such as those described in [27, 28]). Because contextual information is collected across several devices, and must be used by different devices when they are planning, Accord must manage context information in a decentralized manner. Just as there is no central file server for many users, there is also no central repository of usage information.

Since we may need to transfer a file between two devices that are never directly connected, Accord provides support for *multi-hop transfers*. Other devices that connect to both of the file transfer endpoints are used as intermediate steps in the file transfers. Determining which intermediate devices to use, especially given resource constraints on those devices, requires careful planning. Accord implements a planning framework that uses information about usage and connectivity to prioritize certain transfers, making the best use of scarce resources.

The components of our architecture are illustrated in Figure 1. The components in the figure are divided into two groups: the **Accord broker** and **client modules**. Each device runs a single instance of the Accord broker, which coordinates all of the gathering and management of file context information, as well as communicating with brokers on other devices to synchronize the state

of the data-space. As such, the collection of brokers manage global, loosely-synchronized state about user data and context information. The client modules provide stubs and libraries to allow individual applications running on the device to interact with the Accord broker. We describe each of these groups of components in turn.

3.3 Accord broker

The Accord broker is responsible for managing all of the user data-space functionality on a particular device, and communicating with brokers on other devices. The collection of brokers on all devices together provide the total Accord service. A broker interacts with brokers on other devices on a pairwise basis. When two brokers are connected, they periodically synchronize with each other (with frequency determined by a user-specified parameter). If two brokers become disconnected, they synchronize upon reconnection, and then resume periodic synchronizations for as long as they remain connected.

The broker runs as a daemon in the background on a device, and accepts requests from client modules via the **Broker API**. The API uses Java RMI to accept requests. The main thread of control in the broker resides in the **manager**. The manager accepts all requests from the API, and initiates actions at the other broker components to handle the requests. In particular, the manager:

- Updates the context information based on signals from the client modules
- Directs synchronization of state information with brokers on other devices
- Transfers files according to replication plans

The broker updates context information in three modules. The primary information resides in the **object graph**. The object graph represents user files replicated by Accord, as well as metadata and context information about these files. Our goal is to keep the graph itself compact, to minimize storage usage on resource-constrained devices (such as PDAs, cell phones or USB keys). Because the metadata can become quite large, it is managed separately from the structure of the graph. This allows us to store the whole graph on all of the devices, while only storing as much metadata on a device as is relevant and feasible. The object graph is described in detail in Section 4.

The **connectivity graph** stores information about the connectivity between devices. Each device in the user data-space is represented as a vertex in the graph, and edges between vertices are annotated with statistical information about the frequency, duration and bandwidth of connections between those devices. The connectivity graph is used by the replication planner. The **other statistics** module keeps other information used by the planner, such as the frequency with which a user utilizes a particular device.

The **replication planner** generates plans for moving files between devices. A simple replication plan might specify that an important file is replicated to all other devices whenever possible. A more complex plan might target replicating a file to a device that is only intermittently connected, and specify multiple hops to achieve this replication. The replication planner is described in Section 4.

The **rules engine** allows us to specify consistency rules for the management of the context state. For example, we might specify that cycles are not allowed in the object graph.

The communication modules manage the high-level interaction between brokers. Our implementation uses Java RMI for most operations, and a simple line protocol for file transfers. When two brokers synchronize, they first activate their respective **graph and statistics synchronizer** modules to synchronize their context state. This information is used by the replication planner, and therefore must be as up to date as possible on as many devices as possible, so we synchronize it first. Each broker receives a copy of the object graph, connectivity graph and other statistics from the other broker, and updates its local state. The synchronization process is described in detail in Section 4.

The next stage in the synchronization process is to activate the **plan propagator**. When a broker is attempting to execute a multi-hop plan, it must tell the broker on the next device in the plan what the total plan is. Then, the broker receiving the plan can decide whether to continue executing it, or whether to replan based on any new information it has. Similarly, each device must know about active plans so that it can potentially short-circuit the movement of data if it gets the opportunity. For these reasons, the plan propagator copies all *active plans* (that is, plans for files that have not yet reached all of the targeted devices) to each broker it synchronizes with.

The third stage of synchronization is to transfer the files that need to be copied. File transfer is the third stage because transferring any particular file is less important than keeping the global context and plan information synchronized on different devices. The **file replicator** transfers as many files as it can, given connectivity and storage constraints, in order of decreasing priority as determined by the file transfer planner. Note that these replicated files may later be deleted from a device to conserve space, requiring that files be transferred again if they will be needed again.

The **introduction protocol** operates independently of the synchronization between brokers. Whenever a new device is added to the user data-space, it must be "introduced" to the other devices. This involves assigning the device a unique name, transferring the initial copy of the context state to the new device, and initializing other operation parameters. This introduction can be done the first time that the device connects, and requires only a single other device that is already in the data-space. Once the new device is introduced, it can begin synchronizing other devices as a regular member of the data-space. For the special case of the first device added to Accord, the introduction protocol simply initializes the Accord service and assigns a name to that device without waiting to contact other devices.

Dumb devices "Dumb" devices (such as USB keys or iPods) cannot run an instance of the broker. When a dumb device is synchronizing with a smart device, the smart device must perform all of the work. For example, the smart device performs object graph merging for both itself and the dumb device, writes the

object graph, connectivity graph and other statistics to the dumb device, and so on. The dumb device needs all of this context information so that it can give it to any other smart devices it interacts with later. Similarly, if a plan requires a dumb device to "push" a file to the smart device, the smart device will actually have to "pull" the file itself. To implement these smart/dumb interactions, our system includes special modules (called **dumb peer proxies**) that provide the same API as the RMI stubs for smart/smart broker interaction, but which perform writes to the dumb device rather than transfers over RMI.

3.4 Client modules

Individual applications interact with the Accord broker via the client modules. Each application links to an Accord library that provides stubs for performing Accord operations. Each file that is managed by Accord is represented using an **Accord file** object. This object provides methods to open, close, read and write the file, among other operations. In addition, the file object provides methods to create or destroy relationships between files in the object graph, update the file metadata, and perform other Accord-specific operations. We have extended the `java.io.File` class with these Accord operations, and in ongoing work we plan to develop similar libraries for other programming environments.

The **Accord filesystem** manages a collection of Accord files. The filesystem acts as a factory to instantiate Accord file objects whenever an application wishes to access a file managed by Accord. The filesystem object also provides operations to navigate the whole set of files managed by Accord, as well as all of the relationship information between files that Accord tracks.

We have also developed a **legacy filesystem interface**, so that Accord can be accessed as if it were a normal system drive. This allows us to read and write data to Accord using non-Accord-aware applications. To support this interface, the Accord daemon acts as an NFS server, and devices "mount" the Accord filesystem using an NFS client. We chose NFS because of its portability across devices. We have integrated the JNFSD Java open source server [29] with Accord to provide the NFS interface. For performance reasons, the NFS interface resides in the same process/address space as the broker, rather than using Java RMI to access the broker API. (Other applications use RMI because they must perform inter-process communication to talk to the broker.) We are also designing a **graphical user interface** to interact with Accord, although this interface presents HCI challenges that are still the subject of ongoing work. In addition, we have implemented a shell, similar to Unix `sh`, that allows us to interact directly with the system. This shell is an example of an Accord client application.

4 Middleware services for predictive file movement

4.1 Context information

Advanced techniques for inferring user patterns require a large amount of information about user activity. Accord tracks simple usage statistics, such as how

often a file is opened for reading or writing, as well as more complex information, such as whether two files are opened together, whether files are opened more frequently on certain devices, whether files are opened for reading on several devices but opened for writing on only one device, and so on. These statistics are gathered from Accord API open, read, write and close operations called by client modules. Context information will allow machine learning algorithms to determine which files are needed where, which files should be co-located, and so on.

In order to track rich context, Accord must go beyond the simple metadata managed by traditional filesystems. Our approach is to represent context information about files as a graph, called the **object graph**. A graph representation allows us to construct and maintain complex relationships between data objects. Each device updates a local copy of the object graph with usage information, and devices merge their object graphs when they synchronize. Multiple synchronizations allow context information to migrate to all devices, so that transfer planners on different devices have the necessary information.

Object graph model The object graph represents files, metadata about files, and relationships between files. There are three types of vertices in the graph:

- **File Objects (FOs)**: representing a logical file managed by Accord
- **File Descriptor Objects (FiDOs)**: representing an instance of a file on a particular device
- **Relationship Objects (ROs)**: representing a relationship between ROs, FOs or both

When a file is first created, Accord will create a FO representing the file, and a FiDO for each device in the system, flagging the FiDOs that currently have the file. ROs are defined by the system and user applications, allowing Accord to be extensible to represent new types of relationships.

An example fragment of an object graph is shown in Figure 2. This figure shows two logical files, `a.doc` and `b.xls`, represented in the graph by FOs (black circles). Each of these files are stored on three devices: a home PC, a PDA and a laptop. The physical copies of these files on these devices are represented by FiDOs (black squares). Because these two files are frequently accessed together, they are connected to a RO of type "OpenedTogether" (white circle). Each graph object has associated metadata, such as the local path (for a FiDO) or the filename (for a FO).

Each object has an object ID that is globally unique in the user data-space. To generate an object's ID, we take the globally unique name of the device which created the object and concatenate it with the value of a counter local to the device, incremented whenever a new object is created. When two brokers synchronize their graphs, objects created by one broker are copied to the other broker, retaining their unique IDs.

In addition to regular metadata, FiDOs themselves are annotated with three pieces of information: the device represented by the FiDO, a flag indicating

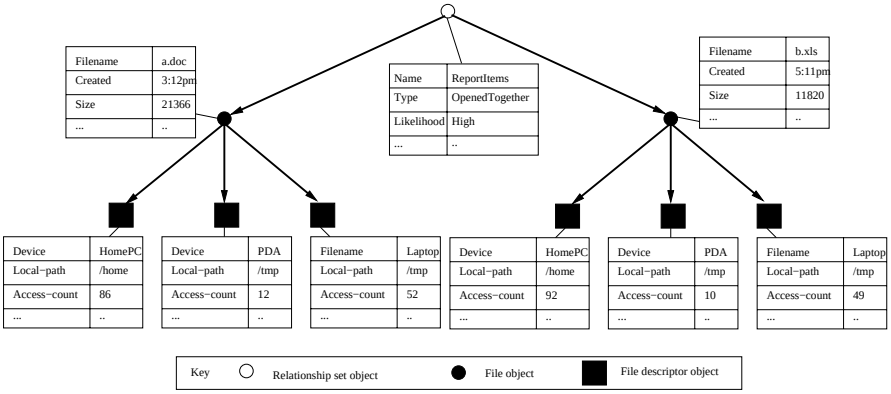


Fig. 2. Object graph fragment.

whether the device currently has the file, and a version vector for the file the device currently has. Version vectors [30] allow us to determine when a new version of a file needs to be replicated to a device that has an older version, and when a potential conflict between two branching versions of a file needs to be reported to the user.

Edges are directed, and may optionally be labeled. Although edges between FOs, ROs and FiDOs could form a general graph, we enforce that FiDOs may only be the children of a single FO, and that FOs may only have FiDOs as their children. ROs can have both ROs and FOs as their children. These constraints allow us to easily maintain simple invariants about the meaning of FOs and FiDOs: we know that a FiDO represents only physical copy information about a particular FO, and that a FO represents only a file, not a relationship.

Graph merging Brokers on different devices update their local object graphs independently. When two brokers synchronize, they merge their graphs by incorporating any changes made on one graph into the other. The result of the merging operation is that both brokers will have identical graphs representing a synthesis of the knowledge each broker had about the system. Existing approaches for distributed state consistency (for example, that operate on atomic values or memory pages) are not directly applicable here, given the special structure and semantics of the object graph.

To synchronize the graph structure, we enumerate structural elements (FOs, ROs, FiDOs or edges) that are present in one graph but not the other. Consider an element X (an object or edge) that is present in graph G_1 but not G_2. If X was never present in G_2, then it is simply added to G_2. However, if X was in G_2 at some point, and then deleted, we must decide whether to add X to G_2 or delete it from G_1. To resolve this issue, we give priority to the most recent information about a structural element. We associate with each element a pair

(Op, TS), where Op is the last operation on the element (create or delete) and TS is the timestamp of that operation. The operation that occurred last wins. For example, if $TS_{\text{create},G_1} > TS_{\text{delete},G_2}$ then the object is created on G_2; if $TS_{\text{create},G_1} < TS_{\text{delete},G_2}$ the object is deleted from G_1. Ties are broken in favor of creation, as a conservative policy. Note that this scheme requires us to retain deleted objects for a while after they are deleted. Currently, we periodically sweep through the system, flushing deleted objects that are older than some threshold. We are examining distributed garbage collection algorithms that would allow us to flush deleted objects as soon as all devices know they are deleted. It is possible that the clocks on the two devices are skewed. However, when the devices synchronize, they can measure the amount of skew between their clocks, and factor in this skew when comparing timestamps.

There are two types of changes that could create a merging conflict:

- An edge connecting two objects was created on one device, but one of those objects was deleted on another device
- Two different FOs (with different IDs) representing the same logical file were created on two different devices

We need mechanisms to resolve these conflicts. To deal with the first conflict, we can either ignore the edge creation or ignore the object deletion, in effect resurrecting the deleted object. One of those choices, say "ignore the edge creation," might be specified as the default, and the other choice used only when a "resurrect" flag is set on the object by the application. To deal with the second issue, we assume that FOs are only created when a new file has been created, and thus different FOs by definition represent different logical files.

It is also necessary to merge metadata. First, note that to keep the graph storage requirements small, a device may store only a portion (or none) of the metadata in the graph. Therefore, we only need to merge metadata items that both devices are interested in. We use a simple "last change wins" rule, as described above for merging edge changes. We also enforce that each metadata item must be retained by at least one device, so that metadata is not lost.

4.2 Adaptive file replication

In order to support the illusion that all files are available on all devices, Accord must transfer new or modified files between devices. Although there may be some benefit to having all files on all devices, storage limitations and connectivity constraints mean that often Accord can only copy a subset of the files to any given device. Our goal is to maximize the number of times that a file needed by a user on a device is already present on that device.

Traditional techniques for replicating information among distributed data stores, such as the epidemic models for database consistency [31], are not sufficient. In particular, we need a method that can prioritize certain transfers over other transfers, or direct a transfer preferentially towards a particular device. Moreover, a file may travel multiple hops over intermediate devices to reach a target device, since not all devices will necessarily be directly connected. If those

intermediate devices have constrained storage, we must carefully plan which files can be transferred using those devices.

We have developed a comprehensive framework for planning multi-hop transfers between devices. Due to space limitations, the complete framework is described elsewhere [32]. Here, we highlight the key aspects of our planning process.

Our replication planner takes as input a set of device-file utilities; each utility represents the value of having a file on a particular device. For example, if a file is always accessed on the user's PDA, then that file has high utility on the PDA. The planner also uses connectivity information about devices to determine the probability that two devices will connect and be able to transfer a particular file. Device-file usage statistics and connectivity information are gathered during the normal operation of the system. The planner will generate a series of plans, representing specific replicas to be made if the broker gets the opportunity; that is, if the broker's device connects to a remote device, and the remote device has enough storage. Each plan p_i has an expected utility, denoted $EU(p_i)$, which is computed from the probability of successfully executing the plan, the device-file utility that would result, and the expected cost of executing the plan. The expression for computing expected utility $EU(p_i)$ is complex and is derived in [32].

Each broker will perform its own planning. For some brokers, this planning process will involve inspecting the object graph and the connectivity graph, and generating an optimal plan. Brokers on resource-constrained devices may be able to perform planning, or may simply accept the plans constructed by others. Dumb devices do no planning, relying on the smart devices to plan for them.

Construction of plans Suppose it is necessary to transfer a file f from a device x to a device y and there are d devices managed by Accord. Our algorithm for constructing plans enumerates all possible plans, and retains only those with positive expected utility:

```
CONSTRUCT-PLANS(f, d, x, y)
1  Set   S ← NIL
2  PlanSet   P ← NIL
3  for i ← 2 to d
4      do S ← S ∪ ( All Plans of length i from x to y )
5  for each plan P_k in S
6      do u ← EU(P_k)
7          if u > 0
8              then P ← P ∪ P_k
9  return P
```

We show in [32] that the plans constructed according to this method will yield the maximum expected utility. Although our algorithm enumerates all plans, our experimental results in Section 5 show that the planning time for a reasonably-sized network of user devices is only a few seconds.

Transfer ordering Once plans have been constructed, they have to be acted upon. Acting on a plan is defined to be transferring the data associated with it and updating its progress.

Suppose a device x has a set of plans S and connects to a device y. Accord will execute the plans in order of decreasing priority, where the priority is defined as the expected utility of executing the next hop in the plan:

```
EXECUTE-TRANSFERS(S, x, y)
1  Set  S' ← NIL
2  for  each P_i in S
3      do if NEXT-HOP-IS(P_i, y)
4            then P_i.value ← EU(P_i)
5                 S' ← S' ∪ P_i
6  SORT-SET-BY-VALUE(S')
7  for  each P_i in S'
8      do TRANSFER(P_i, P_i.data)
```

As we will see in Section 5, this algorithm provides the most benefit compared to epidemic replication when the capacity of any crucial device is limited.

When device capacity is limited, a device must decide which files and associated plans to accept or reject during transfers. In our case, a limited capacity device always accepts newer versions of files it already contains, but when faced with a choice accepts new files only if the plans associated with the new file have higher utility than the lowest utility files already on the device.

5 Results

We have evaluated our system using both an implemented prototype and simulation studies. We have implemented a prototype of Accord in Java, and it runs on Windows, Linux and Macintosh computers. Because it is in Java, it should run on any Java-enabled device (including PDAs and SmartPhones); however, we have not yet completed testing the system with Java 2 Micro Edition for such devices. We have also implemented a simulator for testing our file transfer planner. Using simulations allows us to quickly test a large number of different scenarios and system parameters.

In summary, our results show:

- The I/O overhead of using Accord is roughly comparable to using a remote network filesystem (such as Samba).
- Accord's resource requirements are low. Even for a large user filesystem, the object graph structure requires only a few tens of megabytes. Also, Accord requires minimal time to transfer (8 seconds) and reconcile (800 milliseconds) the graph for a large user filesystem. These overheads scale linearly with the number of files in the system.
- The time to plan increases hyper-exponentially with the number of devices. However, even for a relatively large personal network of 7 devices, construct-

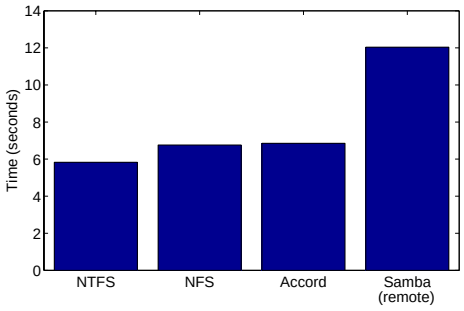


Fig. 3. I/O overhead measured using the Andrew benchmark.

ing plans for 400 files per device required only 2.5 seconds. Usually, Accord will need to plan for far fewer files.

- Planning significantly outperforms epidemic replication, even with only rough usage statistics. In a scenario where a resource-constrained device (such as a cell phone or PDA) was used to transfer files, planning caused up to 34 percent fewer stale reads and 56 percent fewer stale writes than epidemic replication.

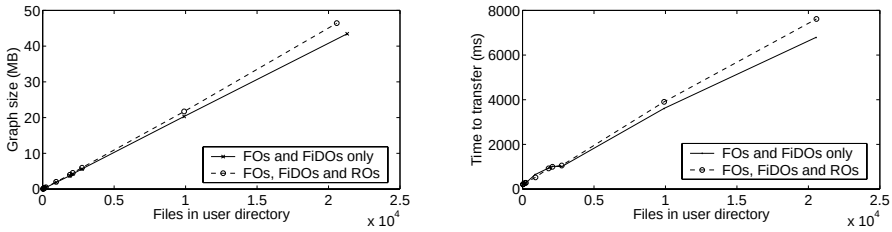
We now describe our experiments and present our results.

5.1 Accord microbenchmarks

We have conducted a number of microbenchmark experiments to evaluate the performance of Accord. Each measurement represents an average over 30 repetitions of the microbenchmark. Our experiments were run on a laptop with 1.6 GHZ Pentium M CPU and 1 GB of RAM running Windows XP. We chose to use a laptop because we wanted to see the performance of Accord on a "typical" user device rather than on a high-power compute server.

I/O overhead Users normally access their files through a native filesystem or via a network filesystem such as NFS or Samba. Accord provides a filesystem interface to allow users to use our middleware as if it were a normal filesystem. Recall that Accord exports the NFS RPC functions and users mount Accord as a normal filesystem using an NFS client. Thus, Accord acts as a networked filesystem, even though it is running on the same machine as the NFS client.

To evaluate the I/O overhead of our filsystem interface, we ran the Andrew benchmark [33] over Accord, as well as three other filesystems: a local, native NTFS filesystem; a local, unmodified NFS server; and a remote Samba server (accessible via 100 Mbit Ethernet). The results are shown in Figure 3. As the figure shows, Accord incurs only 18 percent more latency than a native filesystem. Most of this latency is due to the use of the NFS server; Accord adds only 2 percent overhead compared to a bare JNFSD server. The latency is significantly less than that of a truly remote filesystem, such as Samba.



(a) (b)

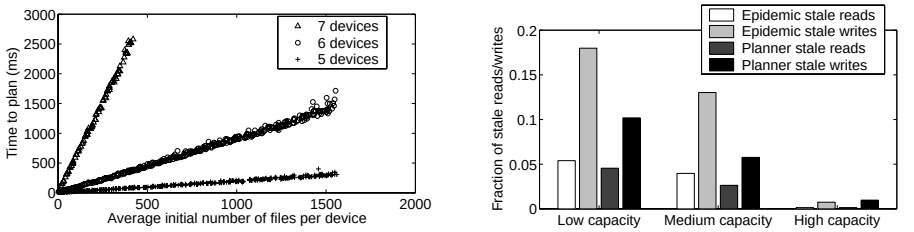
Fig. 4. Object graph: (a) size, and (b) time to transfer between devices.

Graph management Next, we evaluated the performance of Accord's object graph management functions. Accord provides functions for "importing" an existing directory hierarchy into the middleware. This importing process constructs an object graph representing the files in the hierarchy, and informs Accord of its responsibility to manage all of these files. We imported several user filesystems of varying size into Accord to get our measurements. We specified that the user had five devices in his network for these experiments.

First, we measured the size of the resulting object graph. It is important to keep the graph relatively small, so that it can be stored on resource-constrained devices (such as cell phones, PDAs or USB keys). Figure 4(a) shows the size as a function of the number of files in the user directory. The figure shows two data series: one for a "flat" graph, containing only FOs and FiDOs, and one for a graph with ROs representing the user's directory structure in addition to FOs and FiDOs. As the figure shows, even for a large user directory (containing 20,578 files and 20.4 GB of data) the graph is of moderate size, requiring only 43.5 MB of memory. The figure also shows that adding ROs to represent relationships (in this case, directory relationships) requires little extra space, requiring only 7 percent more memory on average. Most of the storage space in the graph is required for FiDOs, since there are five FiDOs created per file (one FiDO per device), and FiDOs are large objects (containing a version vector with a size that scales with the number of devices.) It may be possible to reduce the size of the graph by storing FiDOs more compactly.

We also measured the size of the metadata structure for the graph. Even with very little metadata about each file (its path and filename, read and write count, and last modified time) the metadata store is two thirds the size of the graph itself. Adding more information would only increase the size of the metadata store. Clearly, Accord should manage the graph and metadata separately so that metadata can be ejected to save space on resource-constrained devices.

Next, we measured the time to transfer the graph between devices. We want to minimize this time, in order to maximize the time available to transfer actual files. Figure 4(b) shows the time to transfer a graph between two devices connected by 100 Mbit Ethernet. As the graph shows, the time required scales roughly linearly with the size of the graph. Moreover, even for large user directories, the corresponding graph can be transferred quickly (in under 8 seconds).



(a) (b)

Fig. 5. Planning: (a) time to plan, and (b) planning effectiveness with go-between devices of different capabilities.

Finally, we measured the time to reconcile changes between two graphs. We took the graph for each user directory, and modified it by marking each object in the graph as deleted with probability p. If a RO was not deleted, it was given a new FO/FiDO subtree as a child also with probability p. If a FO was not deleted, it was given a new RO as a parent with probability p. Thus, p controlled the extent of the changes made to the graph. Then, we reconciled this changed graph with the original. The results (not shown) demonstrate that the reconciliation time also increases linearly with the number of user files. The most expensive reconciliation ($p = 0.5$ for a graph of 20,578 files) required less than 800 ms.

5.2 Evaluation of the adaptive file transfer planning

Planning time We ran an experiment to determine the time required to perform transfer planning. We simulated a collection of files stored on a group of devices. For the statistics used in plan construction in this simulation, a pair of devices were modeled as connecting with a probability chosen uniformly at random. Initially, the probability that a given file was on a given device was $1/d$, where d is the number of devices. We then ran our planning algorithm with the goal of replicating every file to every other device. Our planner ran on a 1.7 GHz Pentium 2M, using Java on Windows XP. We allocated 64MB to the Java heap.

Figure 5(a) shows how the time to construct a plan varies depending on the number of devices and the number of files that must be transferred. As the figure shows, for a given number of devices, the time to plan increases linearly with the number of files. As expected, the time increases significantly (in fact, hyper-exponentially) as we increase the number of devices. However, the time is not prohibitive; for 7 devices (a reasonable number for a single user), constructing plans for 400 files per device took only 2.5 seconds. Since plans only need to be created for modified files, usually Accord will have to plan for far fewer than 400 files (as most users rarely update that many files in a day). Our results indicate that it can be feasible to construct the optimal plans, especially if the planning is done only periodically, in the background, and on devices with a large amount of computation power.

Planner effectiveness Next, we ran several experiments to determine how effective the planner was at moving the right files to the right devices. Currently, our system counts the number of file reads and writes on each device, and computes the utility of a file for each device as $writes + reads$. Such a basic formula for utilities attempts to capture a rough estimate of which files are used most often on which devices. In ongoing work, we are investigating machine learning techniques to make better predictions about the utility for each file and device.

We simulated a scenario where a user has two devices at work, two devices at home, and a fifth device that they carry between work and home. The work and home devices might be PCs or laptops, while the device they carry might be a cell phone, PDA or tablet. The user has a firewall at work and home, so the go-between device must act as a courier, carrying files between home and work. We modeled the user as having 1000 files, and using those files over an eight day period. During the first four days, the user works on some subset of his files, and then for the next three days, switches to another subset. Finally, on the eighth day, the user returns to the original subset. This models the case where the user is working on one task (say, a paper), temporarily finishes that task, moving onto another (say, a proposal) before eventually returning to the original task. The subsets of files the user accessed were overlapping, modeling the fact that some files (such as email files) were accessed during all eight days.

Figure 5(b) shows the effectiveness of planning for go-between devices of different capacities: low (20 files), medium (300 files) and high (1000 files). These results represent an average over 20 runs of the simulator for each type of device. The figure shows the number of stale reads and writes, where "stale" means the user tried to read or write a file on a particular device but did not have the most current version on that device. As the figure shows, the planner outperformed simple epidemic replication on the devices with constrained resources. On the low capacity device, the planner caused 16 percent fewer stale reads and 43 percent fewer stale writes, and on the medium capacity device the planner caused 34 percent fewer stale reads and 56 percent fewer stale writes. On the high capacity device, both the planner and the epidemic replication performed roughly equivalently. Clearly, when the go-between device has limited resources, it is important to carefully choose which files to place on the device, and this is what the planner is able to do.

We would like to further minimize stale accesses. We hope that machine learning techniques can help Accord make even better replication decisions. However, even with very rough usage data, the planner is more effective than epidemic replication when devices have constrained resources. Our results are simulated, and represent a particular scenario that is not representative of every user. We are currently preparing a user study to gather actual usage data for our system, enabling us to better evaluate the planner's effectiveness. However, our results already show that the planning approach can be quite effective at moving the right files to the right devices.

6 Conclusions

Our Accord middleware provides system services for emulating a best-effort virtual user data-space. First, Accord implements a loosely synchronized distributed data structure for tracking rich contextual information about user data. Simple usage statistics can be used to make file transfer decisions, while more complex contextual data can be used by machine learning techniques to model user activities as a basis for deciding which files to transfer. Developing such machine learning techniques is the topic of ongoing work that will be supported by our Accord prototype. Second, Accord can plan a robust sequence of file transfers to proactively store data on a device before a user needs it. Such file transfers must take into account variable connectivity between devices and device storage constraints. In particular, file transfers may require multiple hops in order to get to their destination. Experiments and simulations show that the overhead of using Accord is low, and that even with rough usage information Accord can make good file placement decisions.

References

1. Kistler, J.J., Satyanarayanan, M.: Disconnected operation in the Coda file system. ACM TOCS **10** (1992) 3–25
2. Noble, B.D., Satyanayaranan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K.R.: Agile application-aware adaptation for mobility. In: Proc. SOSP. (1997)
3. Reiher, P.L., Heidemann, J.S., Ratner, D., Skinner, G., Popek, G.J.: Resolving file conflicts in the ficus file system. In: Proc. USENIX Summer Technical Conference. (1994) 183–195
4. Huston, L.B., Honeyman, P.: Partially connected operation. Computing Systems **8** (1995) 365–379
5. Edwards, W., Mynatt, E., Peterson, K., Spreitzer, M., Terry, D., Theimer, M.: Designing and implementing asynchronous collaborative applications with Bayou. In: Proc. ACM Symp. on User Interface Software and Technology (UIST). (1997)
6. Liskov, B., Johnson, P., Gruber, R., Shrira, L.: A highly available object repository for use in a heterogeneous distributed system. In: Proc. of the 4th Int'l Workshop on Persistent Objects. (1990)
7. Sobti, S., et al: Segank: A distributed mobile storage system. In: Proc. Third Conference on File and Storage Technologies. (2004)
8. Anderson, T.E., et al: Serverless network file systems. ACM Transactions on Computer Systems **14** (1996) 41–79
9. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proc. SIGCOMM. (2001)
10. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: Proc. SIGCOMM. (2001)
11. Rowstron, A., Druschel, P.: Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In: Proc. SOSP. (2001)
12. Kubiatowicz, J., et al: OceanStore: An architecture for global-scale persistent storage. In: Proc. ASPLOS. (2000)

13. Katz, R.H.: Adaptation and mobility in wireless information systems. IEEE Personal Communications **1** (1996)
14. Roussopoulos, M., Maniatis, P., Swierk, E., Lai, K., Appenzeller, G., Baker, M.: Person-level routing in the Mobile People Architecture. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems. (1999)
15. Zhao, X., Castelluccia, C., Baker, M.: Flexible network support for mobile hosts. Mobile Networks and Applications (MONET) **6** (2001)
16. Baratto, R.A., Potter, S., Su, G., Nieh, J.: MobiDesk: Mobile virtual desktop computing. In: Proc. MobiCom. (2004)
17. Osman, S., Subhraveti, D., Su, G., Nieh, J.: The design and implementation of Zap: A system for migrating computing environments. In: Proc. OSDI. (2002)
18. Kozuch, M., Satyanarayanan, M.: Internet suspend/resume. In: Fourth IEEE Workshop on Mobile Computing Systems and Applications. (2002)
19. Park, V., Corson, S.: A highly adaptive distributed routing algorithm for mobile wireless networks. In: Proc. Infocom. (1997)
20. Broch, J., Maltz, D.A., Johnson, D.B., Hu, Y.C., Jetcheva, J.: A performance comparison of multi-hop wireless ad hoc network routing protocols. In: Proc. MobiCom. (1998)
21. Prasad, S.K., et al: SyD: A middleware testbed for collaborative applications over small heterogeneous devices and data stores. In: Proceedings of ACM/IFIP/USENIX, 5th International Middleware Conference. (2004)
22. Lamming, M., Eldridge, M., Flynn, M., Jones, C., Pendlebury, D.: Satchel: providing access to any document, any time, anywhere. ACM Transactions on Computer-Human Interaction **7** (2000)
23. Davison, B.D., Hirsh, H.: Predicting sequences of user actions. In: Joint AAAI/ICML Workshop on Predicting the Future: AI Approaches to Time Series Analysis. (1998)
24. Horvitz, E., Koch, P., Kadie, C., Jacobs, A.: Coordinate: Probabilistic forecasting of presence and availability. In: Proceedings of the Eighteenth Conference on Uncertainty and Artificial Intelligence. (2002)
25. Dourish, P., Edwards, K., LaMarca, A., Salisbury, M.: Using properties for uniform interaction in the Presto document system. In: Proc. ACM Symp. on User Interface Software and Technology (UIST). (1999)
26. Quan, D., Huynh, D., Karger, D.R.: Haystack: A platform for authoring end user semantic web applications. In: Proc. Int'l Semantic Web Conference. (2003)
27. Omojokun, O., Isbell, C.: Supporting personalized agents in universal appliance interaction. In: Proceedings of the ACM Southeast Conference. (2003)
28. Isbell, C., Omojokun, O., Pierce, J.: From devices to tasks: Automatic task prediction for personalized appliance control. Personal and Ubiquitous Computing **3** (2004) 146–153
29. Mitchell, M.: JNFSD. http://hometown.aol.com/markmitche11/jnfsd.htm (2005)
30. Parker, D.S., et al: Detection of mutual inconsistency in distributed systems. IEEE Transactions on Software Engineering **9** (1983)
31. Agrawal, D., Abbadi, A.E., Steinke, R.C.: Epidemic algorithms in replicated databases. In: Proc. ACM Symposium on Principles of Database Systems. (1997)
32. Roberts, D.L., Bhat, S., Isbell, C., Cooper, B.F., Pierce, J.: A decision-theoretic approach to file consistency in constrained peer-to-peer device networks. Technical report, submitted for publication, available at http://www.cc.gatech.edu/-~robertsd/unidad/unidad-dtp-techrep.pdf (2005)
33. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M.: Scale and performance in a distributed file system. ACM TOCS **6** (1988)