Policy Expressivity in the Anzere Personal Cloud

Oriana Riva*
Microsoft Research, Redmond
oriana.riva@microsoft.com

Qin Yin
Systems Group, ETH Zurich
qyin@inf.ethz.ch

Dejan Juric
Systems Group, ETH Zurich
juricde@gmail.com

Ercan Ucan
Systems Group, ETH Zurich
eucan@inf.ethz.ch

Timothy Roscoe
Systems Group, ETH Zurich
troscoe@inf.ethz.ch

Abstract

We present a technique for partially replicating data items at scale according to expressive policy specifications. Recent projects have addressed the challenge of policy-based replication of personal data (photos, music, etc.) within a network of devices, as an alternative to centralized online services. To date, the policies supported by such systems have been relatively simple, in order to facilitate scaling the policy calculation to large numbers of items.

In this paper, we show how such replication systems can scale while supporting much more expressive policies than previous schemes: item replication expressed as constraints, devices referred to by predicates rather than explicitly named, and replication to storage nodes acquired on-demand from the cloud. These extensions introduce considerable complexity in policy evaluation, but we show a system can scale well by using equivalence classes to reduce the problem space. We validate our approach via deployment on an ensemble of devices (phones, PCs, cloud virtual machines, etc.), and show that it supports rich policies and high data volumes using simulations and real data based on personal usage in our group.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.1.6 [**Software**]: Programming Techniques—*Logic Programming*

General Terms

Algorithms, Design, Experimentation

Keywords

Replication systems, overlay networks, cloud computing, mobile phones

*Work done while the author was at ETH Zurich.

SOCC'11, October 27–28, 2011, Cascais, Portugal.

1. INTRODUCTION

This paper presents a technique for policy-based replication in a network of personal devices, both physical and virtual. We show how to flexibly replicate data in response to a rich set of policies in a way that is robust in the face of devices entering and leaving the system, and with the option to dynamically acquire new resources (virtual machines and cloud storage) in response to changes in workload and policy, if this results in a "better" configuration of the system.

Managing a user's personal data (photos, contacts, music collection, etc.) is a long-standing problem with, as yet, no effective solution [4, 18, 22, 29]. Achieving this without relying on large, online service providers like Facebook or Google is a topic of considerable research interest [24,26,30,31], since a personal approach retains a greater degree of privacy, and is resilient in the face of a provider becoming insolvent, or the victim of a large-scale compromise of private data.

The scenario can be summarized as follows: a user owns a small (fewer than 20) collection of different devices, which might include phones, tablets, laptops, home machines, and virtual machines and associated storage rented from cloud providers such as Amazon. This constitutes the user's *personal cloud*. The user acquires and (less frequently) modifies new data items, by taking photos and videos, downloading music and documents, and editing contacts.

The goal is to preserve this growing body of personal data by replication, and make it selectively available according to the user's applications and needs, specified as a set of replication policies. The problem is complicated by the limited resources on some nodes (such as mobile phones), the bandwidth required to replicate data quickly, and the fact that the set of devices involved can change suddenly (*e.g.*, due to failure, theft, or purchase of new hardware).

Our contributions here are to demonstrate that the range of allowable policies (*i.e.*, the expressivity of the policy language) can be dramatically increased over previous systems without sacrificing scalability. We show (i) how replication policies for personal data can be written independently of specific devices, and can even result in the system acquiring and releasing virtual resources on-demand, (ii) how such a system can react to changes in the environment such as failures or network outages to preserve policy goals, and (iii) how to scale rich policy calculations up to large numbers of data items with only modest requirements in memory and computation, using equivalence classes.

We validate our contributions in *Anzere*, a storage system for personal clouds, integrating personal computers, mobile phones, tablets, and virtual machines dynamically acquired

on both Amazon EC2 and PlanetLab, and evaluate it using personal data from a member of our group.

The rest of the paper is organized as follows. In the next section, we motivate our work and review related work. We then elaborate on our target scenario, and identify the key properties a personal storage system should provide. Section 4 presents our policy model, and a description of the Anzere implementation is given in Section 5. We evaluate the system in Section 6, and conclude in Section 7.

2. BACKGROUND AND MOTIVATION

Automatic personal data management is the motivation for our work. Recent user studies have shown that, despite a plethora of commercial point-solutions for backup and synchronization, people still find it difficult to manage their multiple personal devices [4, 18, 22, 29]. Oulasvirta and Sumari have studied practical problems people face in synchronizing devices [22], for example their laptops can be automatically backed up to a file server, but their smartphones cannot easily access this server. On the other hand, the option of storing all data on smartphones is not always viable due to limited storage capacity. People find ad-hoc solutions to such problems, such as carrying more devices, anticipating future needs by copying data to appropriate locations, and manually synchronizing their data when most convenient (*e.g.*, before leaving for a trip).

Dearman and Pierce [4] report that people synchronize their devices, using portable media, emailing files to themselves, network data sharing, or using third-party external servers [6, 11, 35], but not without the risk of loosing their data as recently reported in the news [1]. All these techniques have serious limitations: they require special configuration, cannot handle all types of files, and/or raise privacy and reliability concerns. File synchronization tools are rarely adopted by regular users partly because they rarely organize their personal data through hierarchical naming, but instead use data attributes [18, 29] and higher-level search interfaces [10, 35, 37].

Motivated by these problems, several recent policy-based replication systems have been proposed. Closest to our work are Perspective [30] and Cimbiosys [26]. Both address the challenge of personal data storage with support for content-based partial replication. Perspective, designed for home devices, provides a semantic file system interface based on the concept of *view*, a query which defines a set of files to be stored on a specific device. Cimbiosys allows users to selectively distribute data across their devices by associating content filters with each device. Devices exchange data and filters through opportunistic peer-to-peer synchronization; the system guarantees filter consistency using an efficient compact log structure for performance. Perspective replicates filters on all devices, and assumes they change infrequently, whereas Cimbiosys allows incomplete knowledge of other replicas and frequently-changing filters using filter-based synchronization trees.

Eyo [31, 32], a storage system for personal media collections, fully replicates policies and all content metadata across all devices – content and metadata are managed separately. The system offers a device-transparent storage API, where each device knows about all objects.

In all these systems, data is selected in policies using logical predicates, but locations are specific devices; in this paper we show how to relax this constraint to devices specified by logical predicates and, indeed, acquired on-demand if necessary. Like Eyo, Anzere fully replicates content metadata and policies, but policies are evaluated by an elected coordinator. We show that this approach deals efficiently with filter changes and adapts to changing network topologies.

EnsemBlue [23], a distributed file system for PCs and consumer electronic devices, provides content-based partial replication through persistent queries, which specify the data an application is interested in receiving. Matching operations on files are logged by the file server in records that can be retrieved by the client. We share with EnsemBlue the device ensemble concept and diversity of the storage elements. We differ in that our goal is to enable data management through device-independent policies and extend the ensemble to dynamically-acquired cloud resources.

PodBase [24, 25] is a system for storage management across a household's personal devices. PodBase's goal is to ensure that data is replicated on enough devices to tolerate failures (data durability) and that replicas are placed on devices where they may be needed (data availability). We share with PodBase the self-managing aspects, and the general goal of automatic data management. PodBase, however, does not focus on providing a policy-based interface for the support of flexible replication requirements. We show that the data durability and availability properties PodBase targets can be expressed by our policy language.

Older systems provide partial replication, but without policies for semantic data management. Coda [14] allows mobile devices to cache files and use hoarding priorities to specify interest, such clients can work disconnected and reconcile later. Coda uses a centralized topology. Ficus [12] and Pangea [28] provide partial replication and topology independence, but without arbitrary consistency guarantees.

In contrast, PRACTI [3] allows applications great freedom to tune consistency, and supports partial replication and topology independence. Our runtime is partly based on the design of PRACTI, which in turn borrows ideas from Bayou [33] and TACT [40]. Above a PRACTI-like replication system, we add semantic data management through replication policies which are integrated in a larger runtime that does overlay management, resource monitoring, and dynamic acquisition of cloud resources.

Many of these systems offer additional functionality in areas orthogonal to those studied in this paper; for example, Cimbiosys provides a powerful access control model [36]. We feel the techniques in this paper are complementary – a practical system would incorporate most, if not all of them.

3. TARGET SCENARIO AND ENVIRONMENT

Anzere is intended for *personal clouds*, personal ensembles of owned machines (mobile devices like phones and tablets, laptops as well as fixed computing devices like PCs) which can be dynamically extended to incorporate also rented cloud resources (*e.g.*, from Amazon EC2), if advantageous for the overall system configuration.

Personal clouds are highly heterogeneous. They consist of both physical and virtual resources, storage capacities range from a few GB to terabytes on cloud resources, processors vary from embedded systems to server-class processors, link speeds range from wireless to 10Gb Ethernet, and pricing

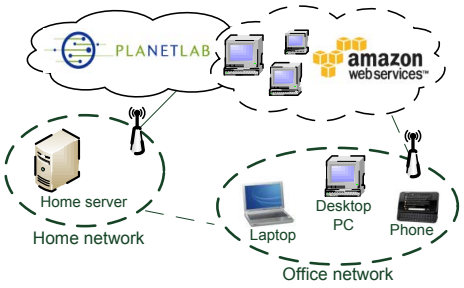


Figure 1: The personal cloud network of our representative user. This is also the hardware setup for our experiments.

structures vary from expensive, metered 3G connections to backbone links.

As a driving example, this paper uses a concrete hardware configuration and a real data set, extrapolating from this where necessary to investigate scaling and tradeoffs.

The hardware configuration, which we use also for our experiments in Section 6, is shown in Figure 1. It comprises an office desktop PC, a home PC server, a laptop, a Nokia N900 smartphone, and a few virtual machines on Amazon EC2 and PlanetLab (the number varies according to the system's policy decisions). The home PC and phone have private IP addresses, and use the virtual machines for NAT traversal as described in Section 5.

The data set, obtained from a member of our research group, is summarized in Table 1. Making strong claims about how representative this data set is would require a full user study and be beyond the scope of this paper, but it does provide us with a starting point grounded in reality.

We see that data is not replicated fully on all devices; instead, partial replicas of data collections are created on different device subsets. In this data set, the music collection has a master copy on the home server, with subsets replicated on laptop, office PC, and phone. For photos, however, the creation of partial replicas does not follow an obvious pattern – there is no master replica, photos on camera are not backed up anywhere else, a subset of the phone's photos is replicated on the laptop. A cloud-based web host stores a (perhaps public) photo subset. The number of videos of this user is relatively small, but this might well increase in the future.

We asked our representative user which goals drive such a data distribution, and gave us the following explanation:

- Backup (as soon as possible) photos as well as videos taken with the camera and the mobile phone on the home PC server. If travelling and the home PC server is not reachable, make a copy of the files on the laptop and ultimately have the laptop synchronized with the home PC.

- Regularly check that enough free storage space is available on his phone and camera. In particular, make sure to empty the camera's memory card when returning from a trip.

- Avoid uploading photos including portraits of himself, relatives, and friends to the cloud. This is achieved by manually checking every photo before uploading.

Table 1: The data distribution (file count and size) of our representative user.

Device	Photos	Music	Videos
HomeServer	6958 (8.1GB)	4904 (23.1GB)	53 (4.7GB)
Laptop	3291 (7.2GB)	932 (5.7GB)	10 (2.2GB)
OfficePC	0 (0)	3997 (19GB)	0 (0)
Phone	89 (38.5MB)	868 (4.3GB)	0 (0)
Camera	25 (56.5MB)	0 (0)	0 (0)
Cloud	4492 (5GB)	0 (0)	28 (435MB)
Total	9231 (13.2GB)	4904 (23.1GB)	56 (6.8GB)

- Store the entire music collection on the home PC server, but have subsets of it on the office PC, laptop, and phone. These collections are selected manually and in an ad-hoc manner, but they tend to include favorite and most recent albums. As the phone's music collection cannot be as big as the one in the laptop and office PC, the user would like this content to be periodically (*e.g.*, weekly, monthly) refreshed such that new content would be always available on his mobile or office devices. The user currently achieves this goal by manually updating the phone's collection, whenever he remembers to do so.

Although this is only one example, it shows a common pattern of use across users and suggests that applications through which users manipulate their data could largely benefit from richer replication policies to control and automate the placement of a user's data across different devices. Rather than the users manually classifying their photos and moving them to the appropriate devices or cloud, an application for generating photo albums, for instance, could use a common face-recognition algorithm to automatically classify any new photo into public or private and replicate it accordingly. Likewise, rather than having users manually refreshing their phone's music collections, a music player application could automatically use user-specific information such as album's rating, creation date, and last played date to periodically refresh the content on the phone.

To support data replication in applications of this type, the goals of Anzere are to:

- efficiently replicate user content according to a flexibly-specified set of policies;

- react in a timely fashion to failures and changes in data, policies, devices, or external conditions such as hosting prices or network outages;

- allow intuitive specification of policies, which are not tied to specific devices, since these change over time – replacing a phone or buying an additional laptop should not require any change in the policies;

- exploit (and decide to acquire or release) dynamic virtual resources on demand, if necessary – that is, Anzere should factor monetary cost, vulnerability, and performance into its replication decisions;

- scale to large numbers of objects, and a reasonable number of policy rules.

The key differences between these goals and those of previous systems are that the policy requirements are considerably broader (variable number of devices, when to rent a new VM, etc.), and devices are, like content, identified implicitly using predicates rather than identifiers. This makes policy reasoning more complex, and in the rest of this paper, we show how to make it tractable in the face of such flexibility. We assume a single-user model: one user owns all replicas in the system.

4. POLICY MODEL

We now describe what can be expressed as policies in Anzere. We do not expect users to directly use our notation; policies are better generated by user applications (*e.g.*, music players, photo sharing applications, etc.) or composed using graphical tools (*e.g.*, a data distribution map across devices), functionality we do not explore here. Instead, our focus is what semantics can be expressed to the system.

We start with the data: a *data item* like a photo is represented as a pair of *(content, metadata)*, where the content is the binary data itself, and the metadata is a list of key-value pairs. Metadata can be mutable, while, for the moment, we assume content is immutable. As explained in Section 5, Anzere also supports mutable content and offers high flexibility in expressing its consistency requirements.

4.1 Device- and content-neutral policies

A *replication policy*, hereafter simply a policy, represents a set of rules, filters, and constraints that applications establish to control where a user's data is stored. A policy might address requirements such as accessibility ("Replicate recently-acquired music on a device the user carries"), durability ("Keep at least 3 distributed copies of the Ph.D. thesis"), privacy ("Do not upload private photos to the cloud"), and capacity limits ("No more than 2GB on phone devices").

Unlike existing systems, policies are independent of the personal cloud for which they are initially specified. To understand the advantage of this approach we consider a simple example. The goal is to guarantee that photos taken with the phone are replicated for durability. Current replication systems (*e.g.*, Cimbiosys, Perspective) express this requirement by generating a filter on the device with id=*myhomedesk* for objects of type=*jpeg* and location=*myapplephone*. In Anzere, the same requirement is expressed through a policy requiring objects of type=*jpeg* and location=*phone* to be replicated to at least one device of type=*fixed* and with tag=*owned*.

The advantage of the second approach is that the replica is not bound to a specific device. The policy can continue to work when the connectivity changes (*myhomedesk* is not reachable by *myapplephone*, but *myofficepc* can be used instead), when the device set changes (the user buys a new phone and calls it *mynokiatablet*) and in principle it can be reused by other users.

4.2 Policy stratification

Anzere policies are sets of triplets <IP,R,DP>, comprising an *item predicate* IP, a *device predicate* DP, and a relationship R that must hold among the items and devices identified by IP and DP. As in other such systems, we find it convenient to express policies in a logic language (Prolog in our case). *Logical unification* is a powerful technique for fusing information from a set of heterogeneous sources, such as different

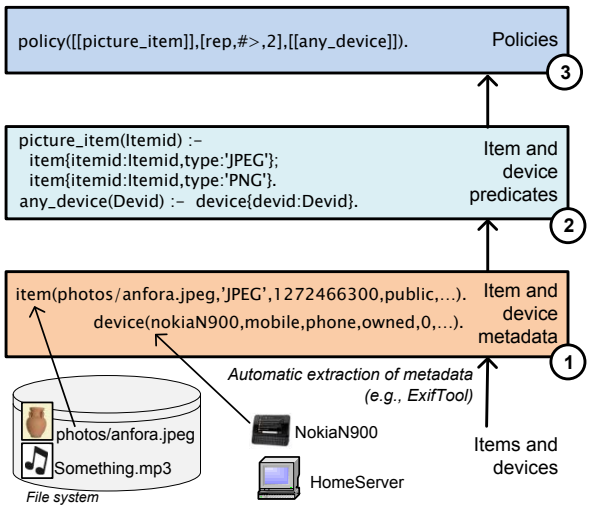


Figure 2: Example of policy stratification. At level 1, facts describing data items and devices are automatically generated from OS and application tools. At level 2, item and device predicates are provided in an Anzere library, maintained by developers. At level 3, user applications use item and device predicates to compose replication policies.

device and data types, and decoupling the system from a predefined schema [34, 38].

To illustrate, consider the example shown in Figure 2. Item and device metadata are automatically extracted from files and the device OS, and represented as Prolog facts. The fact describing the item `photos/anfora.jpeg` says that it is a JPEG photo, was created on 28.4.2010, and is public. The fact for the device `nokiaN900` says it is a phone, a mobile device, owned, and with rental fee of $0.

The example shows how *policy stratification* works. Developers specify item and device predicates, which are applied to low-level facts. Using these predicates, embedded in an Anzere library, user applications can compose policies. Specifically, at layer *2*, the item predicate `IP` is `picture_item` and the device predicate `DP` is `any_device`, defined through simple inference rules – the set of items of type "photo" and the set of available devices, respectively. At layer *3*, a durability policy is defined requiring two replicas of every photo exist in the system; `R` is `rep>2`, which means "replicate to at least 2 devices."

A more interesting example of policy is "make items modified in the last day accessible at no more than 100 ms latency from the phone NokiaN900":

```
mod_item(Op,Time,Itemid) :-
  item{itemid:Itemid,moddate:Moddate},
  mjd_now(MjdNow),
  mjd_to_unix(MjdNow,UnixNow),
  Diff is UnixNow-Moddate,
  Func =.. [Op,Diff,Time], Func.
close_device(MyDevid,Devid,MaxLatency) :-
  ollink{src:MyDevid,dst:Devid,latency:Latency},
  Latency $< MaxLatency.
policy([[mod_item,#<,86400]],[repany],
       [[close_device,'NokiaN900',100]]).
```

`close_device` is a device predicate that selects devices reachable from the specified device within the given latency; it uses `ollink` facts (produced by the monitoring infrastructure described in Section 5.1) which report latency measurements between any two devices in the overlay.

As shown through these examples, predicates can refer to both immutable and mutable properties of an item or device: item category (*picture_item*), device ownership (*owned_device*) as well as time (*mod_item,#<,86400*) and locality (*close_device,'NokiaN900',100*). Multiple predicates can also be specified in a single policy, hence the list syntax for `IP` and `DP` at layer 3 in Figure 2.

Facts describing new types of item and device can be added to the system on-the-fly. If a new pricing model for cloud resources is introduced, or a user's media files are now classified using a new metadata schema, new inference rules can translate between old and new schemas, and the system can immediately start reasoning anew.

4.3 Replication through constraints

A given policy can be applied to a data set and device ensemble in a variety of ways, many of which will be similar in result but only a few of which will be efficient. Consider a camera taking new photographs and two policies: *1: Replicate mobile device files to at least one fixed device*, and *2: Replicate photos to a home device*. If both policies are evaluated in isolation, in order, the camera may unnecessarily upload each photo twice, *e.g.*, to a PC in the office and a server at home. A way to efficiently process the possible policy combinations is needed.

Policies must also incorporate resource and cost constraints imposed by the environment. Mobile phones have limited storage capacity, motivating moving old photos to a better-provisioned device if they are no longer needed. If backup in the cloud incurs a rental fee, a personally owned device might be preferred.

These requirements make this a complex satisfaction problem and we apply *constraint logic programming* (CLP) to it. We are not the first to do this: CLP has been used for system administration [13] and network configuration [5, 20], among other things. CLP programs are regular logic programs with added logical and numeric *constraints*, providing a natural way to express policy requirements such as number of replicas, device capacities, and cost limitations.

The problem consists of finding an allocation of data in the personal cloud which satisfies the constraints. Our formulation in CLP can be visualized as a 2-dimensional matrix `M` whose columns represent the devices and whose rows represent items stored. The variables are all the matrix cells, `v(x,y)`, whose possible values are 0 (do not store) or 1 (store). Each policy imposes constraints on matrix subsets, restricting the possible values that can be assigned to its cells. For example, `policy([[IP]],[repany],[[DP]])` requires submatrix $\mathtt{M}_{(IP,DP)}$=`{(x,y)∈ M:DP(x)=True∧IP(y)=True∧v(x,y)=1}` to satisfy the constraint $|\mathtt{M}_{(IP,DP)}| \geqslant 1$.

A CLP solver computes a set of variable assignments to satisfy the constraints. The matrix solution is then translated into an *execution plan* by comparing the cells of the old and new matrices. Pairs of cells yielding the same value produce no action. Pairs with different values are such that if the new value is 0 a `delete(y,x)` action is generated, and if it is 1 a `copy(y,z,x)` action is generated, where `z` is a device that currently stores item `y`. Responsible nodes then execute these actions.

A CLP solver can not only return a solution to the problem, but also allow for optimization by maximizing a given *objective function*. For instance, an objective function might minimize the *distance* between the current matrix (*i.e.*, the current data placement) and the new solution, where the distance metric is bandwidth utilization. Another might minimize the overall (monetary) cost for renting cloud resources. It is straightforward to add other metrics, and multi-objective optimization is possible using a weighted sum of single objective functions.

Table 2 shows examples which illustrate the expressivity of Anzere's policy language. However, it should be clear from the above description that this approach is unlikely to scale well as the number of objects increases. In Section 5, we show how *equivalence classes* generated dynamically from the policy specification make this approach scalable.

4.4 Acquirable resources

Once we have cast the problem of applying policies efficiently as one of optimizing a set of possible actions, it becomes straightforward to add additional types of actions, with associated costs and benefits, to the basic framework. We exploit this feature to acquire and release cloud storage and computational resources on demand, if doing so results in an overall benefit to the personal cloud ensemble.

Anzere factors the decision of acquiring cloud resources in the policy evaluation itself. The decision is taken entirely on-the-fly by the CLP solver. If the current set of storage devices is not sufficient to satisfy the policy, Anzere searches the possible states the system can achieve by incrementally acquiring resources. Cloud resources simply add new columns to the matrix model described above, the reasoning remains the same. For instance, a scenario we evaluate in Section 6 is that of a synchronizing application which besides synchronizing copies of a user's data also lets the user specify the maximum access time tolerable by specific classes of data. The application uses policies such as policy 4 in Table 2. When the user is travelling abroad, the system detects the increased latency to their home and office network, and can decide to acquire cloud machines from data centers close to the current user's location. On these machines the system establishes temporary copies of data for which the user requested fast access.

The release of cloud resources also occurs in an automatic manner. Cloud VMs are released when no items are stored on them any more, or constraints on rental fees require the content to be copied to cheaper machines. For instance, a policy such as policy 8 of Table 2 would eventually force the system to cleanup unused cloud resources, such as in the example above of the user travelling abroad (when the user returns from the trip and devices re-acquire faster connectivity, cloud VMs are released).

In this context, CLP optimization can be very useful. In fact, the inclusion of cloud infrastructures brings a new interesting variable to the problem: the price for renting VMs and transferring data from/to them. Price constraints are expressed through explicit cost policies or embedded in objective functions. Price models can become complex and need to closely follow the changing pricing structures of different cloud providers. Nevertheless, we feel our approach goes some way to being able to integrate such factors into

Table 2: Examples of Anzere policy expressing requirements such as fault tolerance, data availability, resource management, privacy, and cost for renting cloud VMs.

Policy type	Description	Prolog policy
Fault tolerance:	1. Music backup on a home device	`policy([[audio_item]],[repany],[[home_device]]).`
	2. Video backup on 2 fixed, owned devices	`policy([[video_item]],[rep,#>=,2],[[fixed_device],[owned_device]]).`
Data: Availability	3. 1-day old music on mobile devices	`policy([[audio_item],[mod_item,#<,86400]],[repall],[[mobile_device]]).`
	4. 1-day old photos on a fixed device at 100 ms from the laptop	`policy([[picture_item],[mod_item,#<,86400]],[repany], [[fixed_device],[close_device,'laptop',100]]).`
Resources:	5. 5GB free storage on phone	`policy([[any_item]],[size,#=<,5000000000],[[phone_device]]).`
Privacy:	6. No private items in the cloud	`policy([[private_item]],[repnone],[[cloud_device]]).`
	7. Public photos in the cloud	`policy([[public_item],[picture_item]],[repany],[[cloud_device]]).`
Cost:	8. Rental fee for cloud storage less than 10$	`policy([[any_item]],[cost,#=<,10],[[cloud_device]]).`

the behavior of the system, much as commercial offerings like RightScale [27] attempt to do today for hosted services.

The number of ordinary users today using rented cloud storage for their private files is rather limited. However, Anzere represents a solution also for those users who primarily deal with their own physical devices. More importantly, Anzere can provide an incentive for such users to use cloud storage. By automating the selection, acquisition and release of cloud resources, by offering a simple API for controlling the cost for renting such resources, and by possibly choosing the most adequate cloud infrastructure (based on price and resource requirements) for each user's requirements, the API to cloud computing infrastructures is extremely simplified, thus making cloud computing a viable option also for regular users.

4.5 Composing Anzere policies

We do not expect users to deal with Prolog code. Rather, we anticipate future applications using context-specific knowledge or asking users a few questions to generate most policies on the user's behalf. For example, we have built a prototype photo album application for publishing photos to the cloud, similar to the one our representative user in Section 3 could use for sharing public photos. Users select which photos to include in an album through properties such as size, format, time frame, and privacy, which are automatically extracted from image metadata and using image processing algorithms (*e.g.*, a face recognition algorithm classifies as private photos containing a specific person).

Another example of how existing applications could leverage Anzere's policy language is prefetching. Existing applications (such as web browsers do today by setting their maximum cache size) could integrate support for content prefetching more extensively, and allow users to specify properties such as freshness, overall size, and rating of the content to be prefetched on specific devices. When content is generated on a device in the personal cloud, devices matching the prefetching policy automatically receive such content. For example, our representative user expressed the need to have a means for automatically refreshing music content on the phone. Policy 3 in Table 2 could, for instance, fulfill such a requirement.

Regardless of how policies are specified, conflicts between them may prevent a solution from being found. Two situations can arise: In the first case, one or more policies are issued which conflict with previously-specified policies (*e.g.*, the system requires 2 replicas in the cloud and a new policy specifies that private data cannot be stored in the cloud). The second case occurs when the device ensemble suddenly changes, new data items are submitted or the data properties change such that the current set of policies can no longer be satisfied by the available resources (*e.g.*, the system requires two replicas for all objects within a fixed budget which is exceeded by the generation of new items). Currently, the system reacts simply by reporting detected conflicts and waiting until they are resolved. In a single-user situation, we expect such events to happen in isolation and at a relatively low rate, allowing the feedback returned to the application (and, ultimately, the user) to be accurate enough to quickly identify the cause of the conflict. If such events occur more frequently, a possible solution that we have not yet explored is for the system to use its logs to replay events in isolation and perform a root-cause analysis.

In general, users require a tool to globally manage the policies generated by applications. This can be achieved using graphical tools which can provide an intuitive experience. We think that users should be able to directly observe the "effects" of their policy changes. For instance, if a user adds a policy such as "do not store private photos in the cloud", they should be able to observe that some photos might now have only one or even zero replicas. By understanding the effects of a policy, it becomes easier to achieve the desired result. Our focus in this paper, however, is to explore which policy semantics can be expressed to the system, and how they can be accomplished using a constraint-based approach.

5. IMPLEMENTATION

Anzere is implemented in Python and currently accounts for approximately 32,000 lines of code (counted by David A. Wheeler's "SLOCCount"). The software architecture of Anzere is shown in Figure 3. The main components are the *CLP solver*, including the *knowledge base* (KB) which stores the system's state; a *data replication subsystem* providing flexible consistency and partial replication; and an *overlay network* including *sensors* to monitor the network and device status, and *actuators* to acquire (and release) cloud resources on-the-fly. With the exception of the solver, this code runs on all devices in the user's personal cloud.

Anzere is engineered through a modular framework (inspired by the OSGi [21] module management system), which

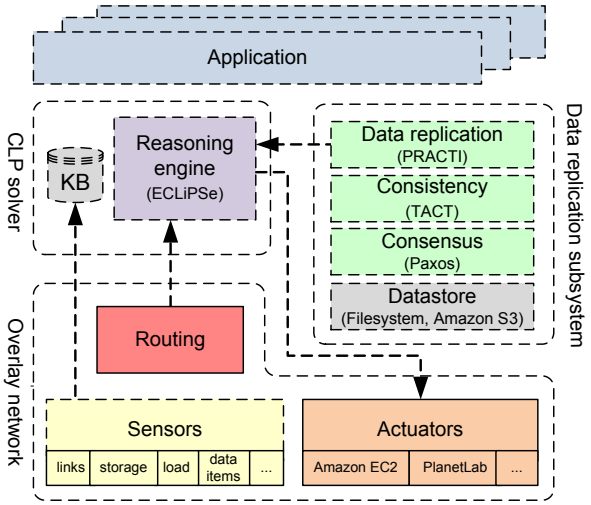


Figure 3: The Anzere system architecture. Main components are the overlay network (sensors, actuators and routing functionality), the CLP soler, and the data replication infrastructure. This software stack runs on each node in the personal cloud network with the exception of the CLP solver which runs only on well-provisioned nodes.

not only enables easy system maintenance, but also allows us to customize the functionality running on each device based on its hardware/OS, as well on the role the node plays in the system (overlay coordinator, member). The CLP solver and knowledge base, for instance, do not run on phones, but are instead located on well-provisioned coordinator nodes. Specialized sensor modules for phones, PlanetLab, Amazon EC2 are included only in the corresponding distributions.

The overlay network of Anzere leverages the Rhizoma [39] infrastructure used by distributed applications running across PlanetLab for resource management. Rhizoma uses a number of sensor modules to monitor the status of PlanetLab nodes, and thus enables applications to dynamically react to changes in load and failures. Anzere reuses Rhizoma's overlay protocols for membership management and failure detection, and PlanetLab-specific sensor and actuator modules, but adds equivalent functionality for personal devices and other cloud infrastructures.

5.1 Sensors and knowledge base

The KB stores a representation of the system's distributed state in the form of Prolog facts. This information is collected through overlay and storage sensors, running on every node in the ensemble.

Overlay sensors monitor the status of the network, detect failures, and collect information such as device type and status, number and type of network interfaces available, latency and bandwidth between any two devices. The main facts carrying this information are:

```
device(devid,location,type,cost,processor,mem,disk).
olnode(hostname,cpuspeed,freecpu,fiveminload,mem,freemem,gbfree).
ollink(src,dst,link-type,latency,bandwidth).
```

Storage sensors inform the KB about policies, item metadata (extracted from files using ExifTool [7]), and so-called *item2dev* information, *i.e.*, a summary of which items are stored at the node. This information is essential for the CLP solver to build a map of the current data distribution and react to changes in policy and new items. In the KB, the facts embedding this information are the following:

```
policy([[item_pred]],[relation],[[device_pred]]).
item(itemid,type,size,createdate,moddate,tag).
item2dev(itemid,devid).
```

Data collected through the overlay sensors is replicated across a few resource-qualified nodes (*i.e.*, nodes that can afford to take over the role of coordinator and run the CLP solver in case of coordinator failures), while storage sensor's information is replicated across all overlay nodes (the overhead is relatively small, *e.g.*, metadata describing 20,000 items accounts for about 6-8 MB).

5.2 CLP solver and equivalence classes

Policy evaluation in Anzere is centralized around one *overlay coordinator* node running the CLP solver. This node is typically a well-provisioned node such as an office or home desktop PC as well as a cloud VM. Phones and tablets are never elected as coordinators. The coordinator does not represent a single point of failure because the overlay is capable of dynamically electing a new coordinator if the old one fails or disconnects. However, a legitimate concern is whether this represents a scaling bottleneck. As we will show in Section 6, this centralized approach has not caused any performance degradation or scalability issues in system operation. Liveness is guaranteed by replicating the KB across all or a few other overlay nodes, to speed up recovering from a lost coordinator.

As CLP solver we use the ECL^{i}PS^e [2] constraint solver, written in C and Prolog. ECL^{i}PS^e is based around a Prolog interpreter with various added libraries for search methods, constraint programming, and interfaces to external solvers.

As the reasoning must be done online, scalability is a primary concern in our CLP implementation. We expect the number of devices in a personal cloud to be small (fewer than 20), and so unlikely to be a scaling problem. Yet, the number of data items may be large, and grows over time. This is potentially an issue for the policy engine, which might need to optimize placement of hundreds of thousands of objects.

To ensure this problem is tractable, we group data items into *equivalence classes*. From the `IPs` present in the active policies, the CLP program directly derives the smallest set of *equivalence relations* which allow the item set to be partitioned into disjoint subsets, namely the equivalence classes. Let us assume a system managing a user's picture, video, and audio items, and two policies:

```
policy([[picture_item][private_item]],[repany],[[owned_device]]).
policy([[audio_item]],[rep,#>=,1],[[fixed_device]]).
```

The program computes four equivalence relations:

```
[[picture_item],[private_item]]
[[picture_item],[public_item]]
[[audio_item]]
[[video_item]]
```

The number and type of equivalence classes as well as the granularity of how data is aggregated into such classes is derived directly from the user's policies, specifically the `IPs` present in the user's active policies. By doing so, the number of classes is much smaller than the number we would

obtain by considering all possible combinations of metadata properties, which would be exponential. Moreover, unlike metadata, `IPs` are boolean predicates, thus ensuring a finite number of combinations.

Having introduced equivalence classes, the matrix model described in the previous section does not refer anymore to item identifiers but instead to equivalence classes. This enormously reduces the number of variables the CLP solver has to manage, and hence its solving time. As we show in the evaluation section, equivalence classes bring a substantial performance improvement to the system.

The solution provided by the CLP solver consists of an action plan for different Anzere nodes to execute. Actions are either *copy* or *delete*. These actions are performed using the replication subsystem's API. If the destination node of an action is a cloud resource which is currently not present in the network overlay, the corresponding cloud actuator is invoked and the node is dynamically acquired and added to the overlay, such that the action can complete. The solver periodically re-calculates the execution plan to incorporate new items and policy variations, as well as to react to changes in the device status and topology (*e.g.*, "device within 100 ms access latency from the phone"). Policies with time relationships (*e.g.*, recently modified items) also require the CLP solver to periodically run – continuous time requirements are approximated. We have not explored yet an event-driven invocation of the solver, *e.g.*, invoking the solver only when substantial changes have occurred in the environment or the action load generated by previous runs is low.

Generating and applying the action plan in an uncertain environment with unreliable network connectivity, where devices can fail or be turned on and off dynamically, requires some care.

First, actions should not be generated every time a condition in the environment changes, otherwise the system may become unstable. For instance, route flapping can cause the network latency to change rather often and mobile devices can disconnect frequently. As explained later, the overlay network smooths these variations such as only stable network changes are processed by the solver. In general, one option we have not yet investigated is to let the user directly trade the responsiveness and stability of the system. Some users might be willing to tolerate a delay in the application of their policies if this implies a more stable system, while other users might prefer the opposite. This trade-off might even be specified on a per-policy or per-data granularity, thus allowing users to prioritize their policy and data treatment.

A second class of challenges concerns the execution of the actions. An action plan might not be fully executed before some of the responsible devices are turned off, and actions happen asynchronously. For instance, in a naïve implementation an item could be deleted by a device before a copy of the same item is executed. To deal with these issues, the coordinator node running the solver needs to maintain a coherent view of the system state. Each time a device completes an action, an acknowledgment is sent to the coordinator (in the form of *item2dev* fact). A *move* action is implemented as a copy followed by a delete. Only after the copy has completed, the coordinator issues a delete. Thus, at any point in time, the coordinator has an up-to-date view of which items are present on which devices. If some devices are turned off before completing their execution plan, the coordinator knows their latest status and can re-issue requests for missing actions, if the devices come back before the next policy evaluation. In this way, unnecessary requests are avoided and policies are enforced in a timely manner.

We ran into cases where thousands of actions were requested together (*e.g.*, when simulating the permanent crash of a home PC) and could not be completed before the next solver invocation, or completed without updating the KB due to delayed acknowledgments. To avoid duplicated action requests, Anzere keeps a queue of pending actions, which are removed from the queue upon completion or timer expiry.

5.3 Data replication subsystem

Anzere uses the replication subsystem to replicate user data as well as information necessary for the system operation (*e.g.*, overlay and storage sensors' information).

For Anzere it was important to support the three PR-AC-TI [3] properties: *Partial Replication* is necessary to address the different resource requirements of a heterogeneous ensemble of devices; *Arbitrary Consistency* permits tuning the cost of consistency when dealing with different types of mutable content; *Topology Independence* is a requirement for the entire system to ensure device failures and mobility do not compromise system operation. Our implementation builds on existing replication techniques, in large part on PRACTI, but also TACT [40], Bayou [33], and Paxos [16].

As in PRACTI, partial replication is achieved through separation of bodies and invalidations. Bodies, stored in the datastore (*e.g.*, file system), contain the actual value of the writes. Invalidations are metadata describing write operations, in the form *<itemID,logical-time>* – *logical-time* is the Lamport clock of the node generating the write. Precise invalidations are sent to a node only if it has subscribed for that item. Imprecise invalidations summarize the information of several writes, and are sent to all nodes in the overlay, thus allowing each node to maintain consistency invariants despite partial replication. To perform read and write operations locally, each node maintains a *log* of received invalidations and stores bodies in a *checkpoint*. Every time an invalidation is received, it is stored in the log and the checkpoint is updated. For instance, if the invalidation reports a write to "item123", the entry "item123" is marked as "invalid" until the new body is received.

Invalidations are exchanged via a causally-ordered stream. A protocol similar to Bayou's log exchange protocol (based on version vectors and logical time) is used to efficiently select a sender's updates which are needed by a receiver. This mechanism ensures that each node's state reflects a *causally consistent* view of the system. To achieve *sequential consistency* Anzere uses Paxos: all nodes in the overlay reach consensus on the total order of all invalidations.

The combination of these protocols provides the basis for a broad range of consistency guarantees. Specifically, consistency can be specified on a per-item basis (at item creation) as well as on a per-read, per-write basis. For example, a blocking read requires the system to first fetch the latest version of the item, while a non-blocking read returns the local copy. Likewise, writes can be performed locally if the consistency requirements associated with the item and the write itself allow, otherwise the latest copy of the item must be first obtained. The system provides a continuous range of consistency levels through the concept of conit-based con-

tinuous consistency of the TACT system. Each item belongs to a *conit*. A conit limits the current order deviation, which is the number of writes that occurred without synchronizing with other nodes. Every time the order deviation exceeds the conit, Paxos is invoked such that the outstanding writes are committed in a sequentially consistent order. In addition to conits, one can specify *bounds* on the number of outstanding invalidations. This conits and invalidation bounds allow an application to tune consistency at runtime.

Using these techniques, copy and delete actions are implemented. Besides copying or deleting the file content on the specified nodes, this involves updating subscriptions and invalidation streams across the interested nodes such that consistency can be guaranteed.

5.4 Overlay network and actuators

The third PRACTI property (topology independence) is achieved using a self-managing overlay network. An overlay node is elected as *coordinator*, while the others act as *members*. The election mechanism can be based on any device attributes (*e.g.*, resource availability, location, owned vs. rented device, etc.). In case of coordinator failure or overlay partition, a new coordinator is re-elected automatically and liveness is guaranteed by replicating the KB across a few overlay nodes.

Although a user's personal cloud is generally small, its complexity arises from the heterogeneity and instability of its resources. Overlay sensors allow the coordinator to detect and react to events such as node failures, variations of link quality, introduction of new network links as well as cloud outages. To avoid flapping in the network measurements, the coordinator maintains a moving average of the last few hours of operation. Devices voluntarily (and temporarily) leaving the overlay directly inform the coordinator, while events such as permanent loss or replacement of a device are reported by the user and are treated differently (*e.g.*, a temporary failure of the home PC is treated differently from a permanent crash).

Paxos also does not restrict the topology. If acceptors detect a proposer's failure or if a new acceptor wants to join the current group configuration (called *view*), a *view-change* is initiated. A group member proposes a new *view-id* on which all members have to agree, and can then form a new view. The protocol ensures that only one view is established at the time [17]. In extreme cases (more than 50% of the devices are simultaneously turned off), the elected coordinator creates a new Paxos view (as done at system bootstrap).

Overlay routing also uses logic to reason about the diverse network and device types and provide path optimization based on latency, bandwidth, and price. For instance, using information about available interface types and node-to-node latency, the routing module can setup a minimum latency path "phone-laptop-cloud" which uses the phone-laptop Bluetooth link and the laptop-cloud Ethernet link. The experiments we present use such mechanisms to automatically connect devices residing in private IP networks.

Finally, actuators are modules that allow the overlay to dynamically add extra cloud resources to the overlay. The decision on when and which resources to add or remove is taken by the CLP solver and communicated to actuators through *acquire* and *remove* actions. Adding a node to the overlay implies copying onto it the code necessary for running Anzere. At startup the node contacts the coordinator

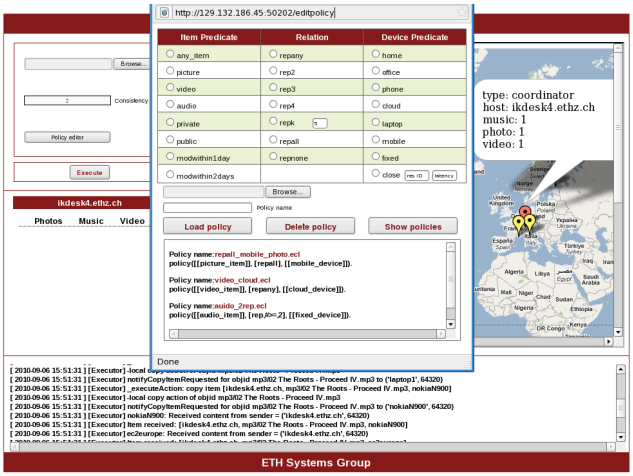


Figure 4: Anzere management interface.

(whose address is provided by the actuator), joins Paxos, and synchronizes with the system state. We currently support actuators for Amazon EC2 and PlanetLab.

5.5 Debugging Anzere

Our current system prototype uses the web interface shown in Figure 4 for debugging purposes. The interface allows developers to keep track of which devices are online, browse or add content, edit their policies, and monitor the solver's execution. In particular, the debugging window shows the output of the CLP solver, copy and delete actions and duration of their execution. This is not by any means intended to be the final UI that ordinary users should be provided with.

6. EVALUATION

We evaluate how Anzere meets the following goals:

- quickly converging to finding a suitable data placement in compliance with the specified policies and reducing the computation complexity using equivalence classes;
- allowing applications tuning of the window of item vulnerability by varying the solving interval;
- reacting promptly to resource disruption and mobility using acquirable resources.

Our target scenario is the one depicted in Section 3: a single user model, with a dataset of 15,000–25,000 data items and a policy sets of 20–30 policies. In the experiments, we use a real personal cloud (shown in Figure 1) emulating the configuration of such a user. The testbed consists of an office desktop PC, home PC, laptop, Nokia N900 smartphone, and cloud resources (two Amazon EC2 VMs, one located in Europe and one in US, and one PlanetLab VM). The usage of cloud resources varies according to policy decisions. The home server and the phone have private IP addresses, but the Anzere routing module takes care of automatically establishing tunnels for these devices using the office PC or cloud machines as traffic forwarders. The phone and laptop use WiFi for communication.

At startup, the office PC acts as the overlay coordinator and runs the CLP solver. All nodes participate in Paxos,

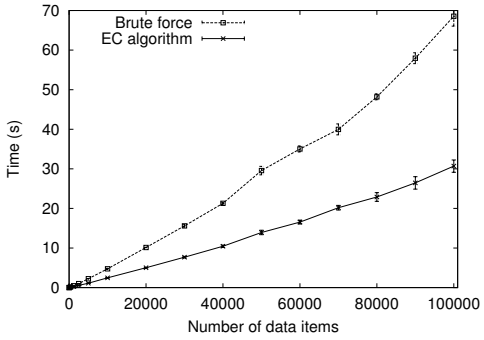


Figure 5: Solving time vs. data set size.

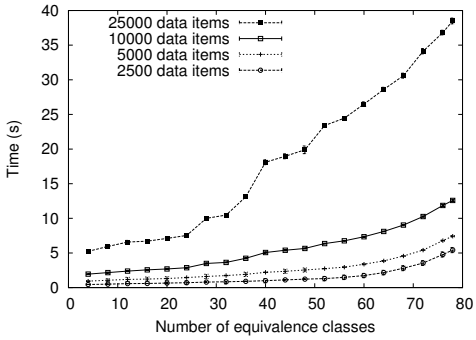


Figure 6: Solving time vs. number of equivalence classes.

and run the overlay sensors collecting information on device status and link performance. In the experiments, we use a workload consisting of real-life samples of photo, music, and video files, with average sizes of 1.1 MB, 3.3 MB, and 4.2 MB, respectively. Metadata files, generated using `ExifTool` [7], are about 300–400 bytes large. We start with evaluating the CLP solver performance and then analyze the full system in operation.

6.1 Policy sustainability

We exercise the CLP solver to study the scalability aspects and resource overhead of policy evaluation. The CLP solver runs on a single well-provisioned cluster node, which has a 2.3GHz AMD Opteron processor and 16GB of RAM. For the tests, we generate a number of data sets directly from the `item` and `item2dev` facts describing the media files of our user's data collection. For smaller data sets, we use random subsets of the original fact list, while for larger ones we randomly duplicate items in the original list, until the desired data set size is reached. Policies used in these tests are similar to those shown in Table 2. We do not report the exact policy set used in each experiment because in terms of solving time all policies are basically equivalent. Instead, what matters is the number of equivalence classes derived from the policy. Graphs report median values and standard deviation based on 20 repetitions with different data sets.

We compare two algorithms: the brute force algorithm, which constrains each item's placement by reasoning item by item, and the equivalence class (EC) algorithm, which generates equivalence classes from the policy set, groups items accordingly, and then solves the placement problem. Figure 5 compares solving time of the two algorithms for an increasing number of items, when 10 policies (corresponding to 12 equivalence classes) are specified. The solving time increases linearly with the number of items, and can handle quickly even a dataset of 100,000 items. If we consider the data collection of our representative user in Table 1 whose size is about 15,000 items, less than 5 seconds are enough to process the entire collection. Our system can scale to handle very large numbers of data items and equivalence classes allow the system to roughly double the speed of the solving process. We see a linear increase for the EC algorithm due to the overhead of expanding the solution matrix (expressed in equivalence classes) into an execution plan consisting of per-item actions. This suggests the performance could be improved further by producing actions based on equivalence classes and delegating the conversion into item identifiers to the devices responsible for their execution.

If the size of the data set does not represent a scalability bottleneck, the other two variables to consider are the number of devices and number of policies. An increase in the number of devices brings a linear increase to the solving time as this corresponds to an increase in the number of columns in the solving matrix. As the number of devices in a personal cloud is likely to be small, we keep the overlay size constant at 7 nodes. Instead, an increase in the number of policies, hence equivalence classes, can cause an exponential growth in solving time. When varying the number of equivalence classes from 4 to 78 (corresponding to 4 and 43 policies) and measuring the solving time for data sets of tens of thousands of items, we found Anzere still executes reasonably fast and with stable behavior. Results are reported in Figure 6. Note that the observation above about delegating to overlay nodes the task of converting the solution from equivalence classes to per-item actions also applies here, and could reduce the performance gap between the data sets.

These results show how the current implementation is capable of fully supporting our target scenario of a single user with a collection of roughly 20,000 data items and 30 policies. The only constraint that remains to consider is the resource overhead of the ECL^{i}PS^e solver.

Figure 7 depicts the solver's upper-bound memory consumption, defined as the total heap space and sum of four different stack peaks (storing Prolog variables, backtracking information, checkpoints, etc.). The peak value gives the maximum allocated during the session. For a data set of 10,000 items, the upper-bound memory usage is within 64 MB, and even with 100,000 items, the upper bound is still within 300 MB. These are more than acceptable requirements for current mainstream personal computers. Recall that the overlay coordinator and the CLP solver always run on well-provisioned nodes, not on phones or tablets.

6.2 System resource overhead

Anzere leverages and combines existing techniques for data replication and couples their execution with resource management. It is reasonable to ask whether this approach may be prohibitively expensive.

We measure the message overhead of the system in a standard scenario where the office PC produces a new photo every 10 seconds and replicates it across 7 devices in the overlay. While doing this, we decrease the level of consis-

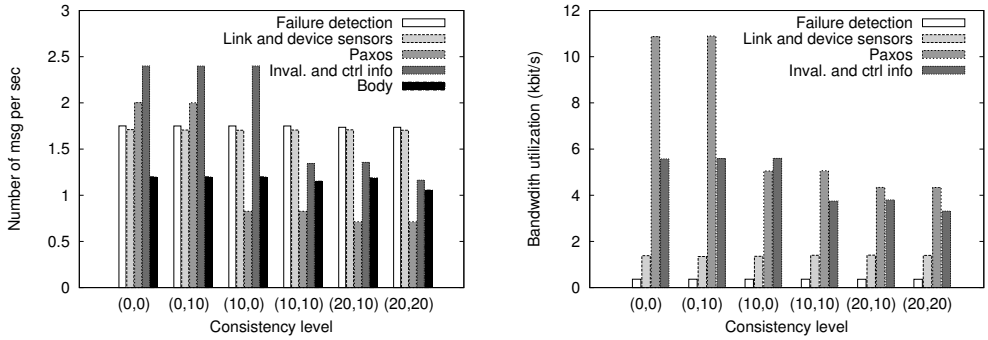


Figure 8: Message overhead at the proposer node for different levels of consistency specified as (conit, send bound).

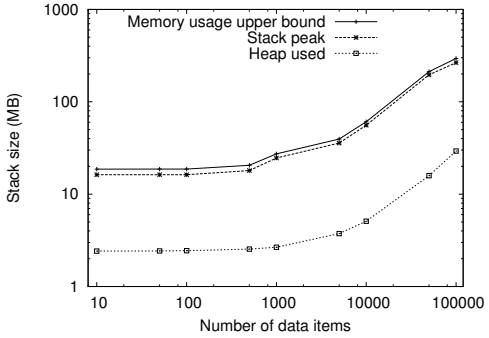


Figure 7: CLP memory consumption.

tency of such data items, and execute 50 writes with each setting. A consistency level is specified by the tuple `(c,s)`, where `c` is the conit bound (*i.e.*, maximum number of outstanding writes) and `s` is the send bound (*i.e.*, maximum number of outstanding invalidations). The smaller `c` and `s`, the stronger the consistency.

For each new write generated at the office PC node (also acting as Paxos proposer), if the conit (or send) bound is exceeded, the node sends all outstanding writes (or invalidations) to all other replicas in the overlay. Figure 8 shows the number of messages sent by the Paxos proposer in each consistency scenario (on the left), and the bandwidth consumed (on the right). We do not plot the traffic due to body exchange as the size of bodies is obviously much larger and application-dependent. For a Paxos acceptor the number of messages is slightly smaller.

As the consistency guarantees become weaker, the overhead of Paxos and invalidation messages decreases. In scenario (0,10), for instance, the number of Paxos messages is roughly twice the number exchanged in (10,10). Likewise, the number of invalidations decreases from scenario (10,0) to (10,10). We also observe that the send bound parameter has an impact on the message overhead as long as it is lower than the conit bound (*i.e.*, when consensus is invoked, invalidations must be exchanged regardless of the send bound). This is the reason why the first two scenarios are identical.

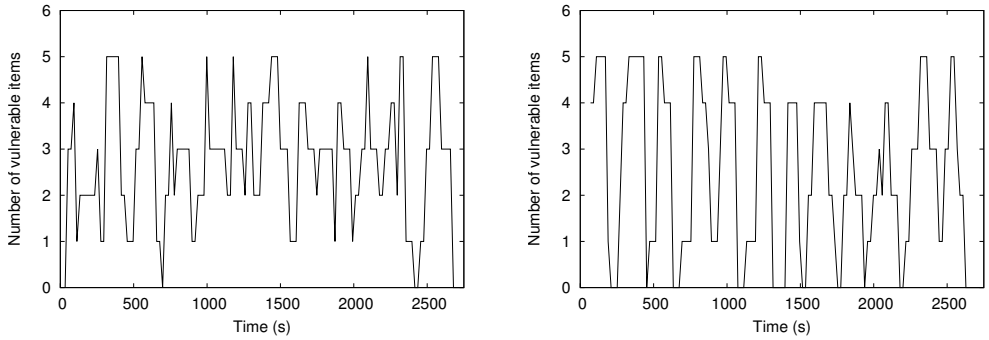
These experiments do not claim any extraordinary result, but serve to calibrate our system, and show that Anzere's message overhead is largely affordable for a modern personal cloud. The number of overlay messages for failure detection and for monitoring network links and devices is constant, and costs about 2 kbit/s. Interestingly, Paxos also appears to be reasonably cheap. Even in the scenario (0,0), with strongest consistency, it costs only 11 kbit/s. Running consensus 730 times a day across a user's personal ensemble consumes about 1 MB of data. This makes Paxos affordable also for less-powerful mobile devices. One round of Paxos across the 7 devices of our testbed takes about 2 seconds.

6.3 Steady-state tradeoffs

The CLP solver has proven fast and cheap to run. The next question is how often it should run and which tradeoffs are involved. We consider a scenario in the steady-state behavior of the system. Every 2 minutes, the Nokia N900 phone generates blocks of 5 photos, 20 seconds apart. Every 20 seconds, the phone (and all other devices) send *item2dev* reports to the KB, thus informing it about the new photos. Photo metadata is fully replicated. Photo bodies are replicated according to 8 policies (12 *ec*), including policies 6 and 7 of Table 2 and `policy([[private_item]],[repany],[[fixed_device]])`. With this policy and device configuration, the solver's solution is to copy private photos to the office PC and public photos to a cloud VM.

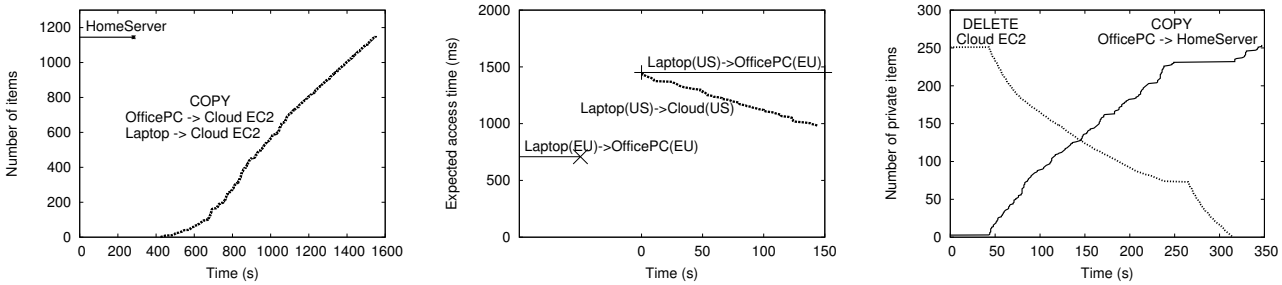
Figure 9 shows the impact the solving period has on the number of vulnerable items in the system. Item vulnerability is defined as the number of items not yet replicated to achieve a stable state – compliant with the policies. Peaks appear when new items are generated in the system, and disappear once copy actions have completed. The average vulnerability decreases as the solver runs more frequently. As shown, with a solving period of 1 minute the level of vulnerability reaches 0 once all copy actions are executed, while with a period of 2 minutes 0 is reached more rarely. In an implementation in which the CLP solver was invoked in an event-based manner rather than periodically, the delay between the first event and the solver's invocation could be adjusted for a similar tradeoff in resource overhead versus data vulnerability.

The rate at which devices report the list of their stored items (based on which the solver takes actions) is not a limiting factor for these scenarios. In fact, transferring *item2dev* reports has such a low overhead that it is possible to set the reporting period to be very short (*e.g.*, 2 seconds) or simply let each device report its new items as soon as they are generated. The limiting factor here is clearly the bandwidth



(a) Solving period=120s (b) Solving period=60s

Figure 9: Item vulnerability vs. solving interval (Nokia N900 generating a new photo every 20 seconds).



(a) Crash of the home server (b) User mobility (c) Change in privacy policy

Figure 10: System reactivity.

of the phone's WiFi connection: the phone cannot complete the transfer to the office PC and to the cloud VM before new items appear.

6.4 Reactivity

Next we show how Anzere can autonomously react and recover from node failures, user mobility, and policy changes.

In the first test, we pre-populate the system with about 6,500 media files distributed across all personal devices. Then we permanently crash the home server, storing roughly 1,200 files, thus making the system vulnerable. The remaining personal devices are insufficient to create enough replicas to satisfy the given durability policy. Anzere reasons about how to recover and reestablish a stable state. As Figure 10(a) shows, it autonomously acquires a VM from EC2 and creates the missing replicas (roughly 1,200 files, for a total of 1.5 GB). The files are copied to the EC2 VM partly from the office PC and partly from the laptop. This process successfully stabilizes the system and decreases the risk of data loss to an acceptable level in a fully automated manner.

In the second test, we emulate a case of user mobility. We assume the laptop initially residing in the office network (in Europe), leaves it to later reappear in the US. In the US the laptop initiates a read workload – downloading about 70 recently-created photos from the office PC in Europe, or roughly 100 MB. This could be the case of a user living in Europe and traveling to the US for some time. We evaluate the expected duration of such a read workload, when policy 4 of Table 2 is enabled at runtime.

The policy requires recent photos to be copied close to the user's laptop for fast access. When the laptop appears in the US, the solver reacts by acquiring a local cloud VM and copying the photos to it. Figure 10(b) shows the access time (per photo) achieved by the user before and after traveling to the US; for comparison we also show what would have been achieved by remotely accessing the office PC (in Europe) from the US. This simple test demonstrates the power of our policy architecture. CLP allows for almost any conceivable policy to be specified, and can be used for long-term data placement, but also for short-term events, such as downloading bulk data or leaving the home network. The performance improvements achievable are clear. In our test, we used a laptop with WiFi connection for simplicity, but for a mobile phone with limited 3G connectivity, the performance gain would be even higher.

Anzere is also designed to deal with changes in policy. To show this, we use a collection of 1,000 photos (of which 25% are private) and store them on a cloud VM and the office PC. Policies 6 and 7 of Table 2 are disabled, while `policy([[private_item]],[rep,#>=,2], [[fixed_device]])` and `policy([[picture_item]], [repany],[[cloud_device]])` are active. At runtime we change the policy set by enabling policies 6 and 7 and disabling the last policy. This change causes the system to delete all private photos from the cloud, but also replicate the content previously stored on the cloud to the home server, as the policy set requires at least two replicas of private items on fixed devices. The solver generates roughly 500 actions. As Figure 10(c) shows, photos are deleted from the cloud and at the same

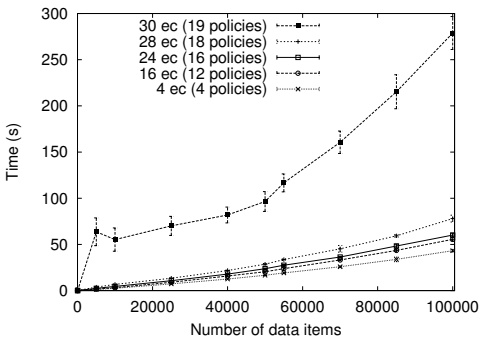


Figure 11: Optimization cost.

time, the same set of photos is copied to the home server (from the office PC). The system automatically reacts to the policy change and reestablishes a stable state.

6.5 Optimization

Finally, we evaluate the cost of supporting policy optimization. As mentioned in Section 4.3, the CLP solver can optionally include a user-defined objective function and find the data placement that optimizes this function. Here we consider the case in which the solver minimizes the bandwidth necessary for enforcing the policy set. We found that the system scales well up to 30 equivalence classes, but after that the solving time can be over 100 seconds for large data sets (results in Figure 11). This suggests using optimization is feasible only with small policy sets, and its overhead is justified only if the cost metric varies considerably across the possible solutions. Further work is necessary before optimization can be made an integral part of the system. In the current implementation, the optimization bottleneck is the ECL^{i}PS^e's `branch_and_bound` search library, which has shown to become very slow with more than 100 variables (corresponding to roughly 32 *ec*). Modern solvers are capable of handling a much higher number of variables, in the order of 1000 variables, and could therefore improve the search time. Alternatively, the search algorithm could be replaced by our own customized implementation.

6.6 Summary

Anzere is a complete platform and has been evaluated in a real personal cloud consisting of phones, laptops, PCs, and cloud VMs. Anzere is capable of autonomously reacting to device failures, policy changes, and user mobility, and we have shown how it can scale to support our target scenarios and the performance gain equivalence classes bring. The system overhead for overlay and device monitoring, and for running the constraint solver was also found reasonable. Yet, our Prolog implementation is completely unoptimized code. In an optimized implementation, the entire policy calculation could be based on equivalence classes thus making the system scale better. For instance, the conversion from equivalence classes to per-item actions could be outsourced from the Prolog base to the actuating devices. The solver itself could be replaced by a more powerful one. We chose ECL^{i}PS^e because of its extensive library support and ease of use, but its performance is not as good as more recent solvers. In Kotthoff's comparative study [15] with three other constraint solvers, ECL^{i}PS^e is largely outperformed by Gecode [8] and Minion [9]. Microsoft Solver Foundation [19] is promising too, supporting programming problems with more than 1000 variables and constraints.

7. CONCLUSION

In this paper, we explore the features that a storage system for personal clouds needs to support, and describe the Anzere prototype that incorporates them. Anzere is novel in several aspects. It supports device-neutral policies, and is thus capable of operating with a changing set of devices. It shows an alternative way to use cloud infrastructure, where cloud VMs are seen as acquirable resources about which the system can reason, and acquire on-the-fly. Anzere not only offers greater policy expressivity, but also a tractable approach, scaling to many data items. Anzere is a novel yet practical system: we have shown it in action in trials across phones, laptops, desktop PCs, EC2 and PlanetLab. Its source code is available at http://www.systems.ethz.ch/research/projects/anzere.

We have so far considered a single-user model. To handle a multi-user scenario, the system can include attributes like ownership, level of trust and owner's relationships in the device representation, without any need to change the reasoning framework. Ultimately, the device space might make use of equivalence classes as well. With the framework in place, extending Anzere to deal with other content type is mostly straightforward. In this paper, we have evaluated Anzere mainly with immutable data, but the replication infrastructure already supports mutable content such as contact information, calendar entries, or text files.

8. ACKNOWLEDGMENTS

We thank Adrian Schuepbach for his help with the Prolog implementation of the policy solving algorithm and Robert Grandl for implementing the Anzere debugging interface. We also thank Andrew Baumann and Eno Thereska for providing valuable feedback and the anonymous reviewers for their insightful comments. This work was partly supported by the ETH fellowship program.

9. REFERENCES

[1] Amazon: Some data won't be recovered after cloud outage. http://www.theregister.co.uk/2011/04/26/amazon_says_some_volumes_lost_in_cloud_outage_not_recoverable/.

[2] K. R. Apt and M. G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[3] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI Replication. In *Proceedings of the 3rd symposium on Networked Systems Design & Implementation (NSDI '06)*, pages 5–5. USENIX Association, 2006.

[4] D. Dearman and J. S. Pierce. It's on my other computer!: computing with multiple devices. In *Proceedings of the 26th international conference on Human factors in computing systems (CHI '08)*, pages 767–776. ACM, 2008.

[5] T. Delaet, P. Anderson, and W. Joosen. Managing real-world system configurations with constraints. In

Proceedings of the 7th International Conference on Networking (ICN '08), pages 594–601, April 13–18 2008.

[6] DropBox. http://www.dropbox.com.

[7] ExifTool. http://www.sno.phy.queensu.ca/~phil/exiftool.

[8] Gecode. http://www.gecode.org.

[9] I. P. Gent, C. Jefferson, and I. Miguel. MINION: A Fast, Scalable, Constraint Solver. In *Proceedings of ECAI '06*, pages 98–102, 2006.

[10] Google desktop. http://desktop.google.com.

[11] Google Docs. http://www.docs.google.com.

[12] R. G. Guy. *FICUS: a very large scale reliable distributed file system*. PhD thesis, University of California, Los Angeles, June 1992.

[13] IBM. Tivoli enterprise console rule developers guide, Aug 2003. 1st edition. SC32-1234-00.

[14] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM TOCS*, 10(1):3–25, 1992.

[15] L. Kotthoff. Constraint solvers: An empirical evaluation of design decisions. CIRCA preprint 2009/7, University of St Andrews, 2009. http://www-circa.mcs.st-and.ac.uk/Preprints/solver-design.pdf.

[16] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, 1998.

[17] D. Mazières. Paxos Made Practical. Technical report, 2007. http://www.scs.stanford.edu/~dm/home/papers/.

[18] 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 *Proceedings of the 28th international conference on Human factors in computing systems (CHI '10)*, pages 645–654. ACM, 2010.

[19] Microsoft Solver Foundation. http://www.solverfoundation.com.

[20] S. Narain. Network configuration management via model finding. In *Proceedings of the 19th conference on Large Installation System Administration Conference (LISA '05)*, pages 15–15. USENIX Association, 2005.

[21] OSGi Alliance. *OSGi Service Platform, Core Specification Release 4, Version 4.1, Draft*, 2007.

[22] A. Oulasvirta and L. Sumari. Mobile kits and laptop trays: managing multiple devices in mobile information work. In *Proceedings of the 25th international conference on Human factors in computing systems (CHI '07)*, pages 1127–1136. ACM, 2007.

[23] D. Peek and J. Flinn. EnsemBlue: integrating distributed storage and consumer electronics. In *Proceedings of OSDI '06*, pages 219–232, 2006.

[24] A. Post, P. Kuznetsov, and P. Druschel. PodBase: transparent storage management for personal devices. In *Proceedings of the 7th international workshop on Peer-to-Peer systems (IPTPS '08)*, pages 1–1. USENIX Association, February 2008.

[25] 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 (USENIXATC '10)*. USENIX Association, June 23–25 2011.

[26] 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 *Proceedings of the 6th USENIX symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 261–276. USENIX Association, 2009.

[27] RightScale. http://www.rightscale.com.

[28] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *SIGOPS Oper. Syst. Rev.*, 36(SI):15–30, 2002.

[29] B. Salmon. *Putting Home Data Management into Perspective*. PhD thesis, Carnegie Mellon University, August 17 2009.

[30] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *Proceedings of 7th USENIX Conference on File and Storage Technologies (FAST '09)*, pages 167–182, February 24–27 2009.

[31] J. Strauss, C. Lesniewski-Laas, J. M. Paluska, B. Ford, R. Morris, and F. Kaashoek. Device-Transparency: a New Model for Mobile Storage. In *Proceedings of HotStorage '09*, October 2009.

[32] 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 (USENIXATC '10)*. USENIX Association, June 23–25 2011.

[33] 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 *Proceedings of the 15th ACM symposium on Operating systems principles (SOSP '95)*, pages 172–182. ACM, 1995.

[34] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: an Information Plane for networked systems. *SIGCOMM Comput. Comm. Rev.*, 34(1):15–20, 2004.

[35] Windows Live. http://explore.live.com.

[36] T. Wobber, T. L. Rodeheffer, and D. B. Terry. Policy-based access control for weakly consistent replication. In *Proceedings of the 5th European conference on Computer systems (EuroSys '10)*, pages 293–306. ACM, April 2010.

[37] Xsan. http://www.apple.com/xsan.

[38] Q. Yin, J. Cappos, A. Baumann, and T. Roscoe. Dependable Self-Hosting Distributed Systems Using Constraints. In *Proceedings of 4th Workshop on Hot Topics in Systems Dependability*, 2008.

[39] Q. Yin, A. Schüpbach, J. Cappos, A. Baumann, and T. Roscoe. Rhizoma: A Runtime for Self-deploying, Self-managing Overlays. In *Proceedings of Middleware '09*, volume 5896 of *LNCS*, pages 184–204, 2009.

[40] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM TOCS*, 20(3):239–282, 2002.