Online migration for geo-distributed storage systems

Nguyen Tran* Marcos K. Aguilera Mahesh Balakrishnan
Microsoft Research Silicon Valley

Abstract

We consider the problem of migrating user data between data centers. We introduce *distributed storage overlays*, a simple abstraction that represents data as stacked layers in different places. Overlays can be readily used to cache data objects, migrate these caches, and migrate the home of data objects. We implement overlays as part of a key-value object store called Nomad, designed to span many data centers. Using Nomad, we compare overlays against common migration approaches and show that overlays are more flexible and impose less overhead. To drive migration decisions, we propose policies for predicting the location of future accesses, focusing on a web mail application. We evaluate the migration policies using real traces of user activity from Hotmail.

1 Introduction

Internet web applications are increasingly important to our everyday lives, as we rely on them for email, searching, online storage, online calling, and much more. These applications face a data scalability challenge that is getting worse, for two reasons. First, there is a growing number of users in an increasing number of regions. And second, the storage needs *per user* are growing as more applications become available online, users accumulate more data, and systems collect more information from users to target ads and personalize their experience. As a result, these applications need to be *geo-distributed*, which means they are deployed across multiple data centers around the world, due to constraints on the size, bandwidth, and power consumption of a single data center. Besides providing scalability, geo-distribution also allows a user to be served from a nearby data center, thereby reducing user response times and bandwidth consumption. For that, the user's data should be at the right data center, namely, a data center close to the user. This is called *access locality*.

Unfortunately, data is not always where it should be: users relocate, and the load at data centers becomes unbalanced due to new applications, new data centers, and changes in the network topologies. In these cases, user data needs to be migrated from one location to another; migration is essential to provide access locality and to balance load. This paper considers the problem of migrating data across data centers. We propose a simple abstraction called *distributed data overlay* or *overlay* in short[1], which represents data as stacked layers stored in different places. Overlays are a flexible way to support data migration; they can be used to cache data at remote data centers, migrate these caches from one data center to another, and migrate the *home* of a data object—the data center where the object is stored when it is not cached. If data is replicated across data centers, overlays can be used to migrate individual replicas.

With overlays, migration can be performed *online*, that is, while the data is accessible to users. This is important for three reasons. First, user data can be massive and the bandwidth across data centers is limited, so that migration can take a long time and we do not wish to disable the user account during migration. Second, we want to migrate data opportunistically in the background, using possibly small amounts of left-over bandwidth. This is so because large companies such as Microsoft pay for private links with fixed bandwidth to connect data centers, which means that unused bandwidth is wasted money. Third, the policies of when to migrate data can be complex, and we do not want to complicate them further with constraints and predictions of when users will access their data.

Online migration is challenging due to races; it requires careful coordination as clients in the network read and write data while the migration process copies the data and the system possibly creates, flushes, and removes caches at remote locations. Overlays are an easy, flexible, and efficient way to handle this coordination, as we demonstrate in this paper.

We implement overlays in a system called Nomad, which is a key-value object store that supports online migration. Key-value stores were recently proposed to support large-scale applications in data centers (e.g., [16]). Though Nomad is a key-value store, overlays are applicable to other types of storage, such as distributed linear-address stores [9], block stores [31], and file systems.

We evaluate the mechanism for migration using a wide-area deployment on five data centers around the world. Our experiments show that overlays impose a small overhead and provide flexibility for supporting caching and migration. They also show that overlay-based migration is more efficient than existing methods based on data locking and logging.

The mechanism for migration is independent of the policies used to trigger migration. We study some simple policies that track the location of users as they move. We evaluate these policies using real traces of user accesses; we compare policies based on access count, time,

*Current affiliation: New York University
[1]not to be confused with a network overlay.

and rate, and we show that, although they are all reasonable, the one based on count performs the best.

Summary of contributions. We consider the problem of building distributed storage systems deployed over many data centers, with support for flexible online migration of data across data centers. We propose distributed data overlays, a simple but flexible abstraction designed to hide the complex distributed protocols (which we provide) required to coordinate access to data at many locations. We also propose policies for driving the migration of user data, and evaluate them using real traces from Hotmail. We implement overlays to produce the Nomad system, and use it to compare our approach against less flexible but common alternatives for storage migration.

2 Background

There are many data centers around the world, each with thousands or more machines, subject to crash failures. We do not consider Byzantine failures in this paper. We target a setting where partitions across data centers are rare in the absence of disasters. This can be achieved by connecting data centers via private leased lines with high availability [1, 2]; by using redundant links to maintain operation during planned link downtime (e.g., using a ring topology across data centers); and by routing traffic via the Internet should all the redundant links become unavailable. The data centers run web applications that store user data, such as these:

Application	User data
web mail	emails
web phone	voice mails
web storage	personal files
chat	text/image history
search	search history
ALL	profile, activity logs

This data should often be stored at a data center closest to the user, where the user logs into. Migration refers to moving the data from one data center to another—to improve access locality or to balance load across data centers. *Online* migration means that, during migration, the data remains accessible to the applications.

We now illustrate some migration use cases with three scenarios; we later explain overlays and how they support these use cases. In these scenarios, "user data" refers to the data specific to a user that an application needs to serve that user. For example, it could be the user's emails.

Scenario 1 (A long trip to China): A French user goes to China. After several days, the system starts to migrate her user data to China. If she goes back, the French copy is updated with any changes made in China. If she stays in China longer, all her data is migrated and the French copy is deleted.

Scenario 2 (Backpacking in Asia): The French user makes a short trip to China and, soon after, the system creates a cache at a data center in China containing her recent user data (e.g., recent emails). She then travels to Russia, and so the system migrates the cache in China to a data center in Russia. She stays in Russia for some time, and so the system starts to migrate her data from France to Russia, which takes several days. Before the migration is over, she returns to France, so the system applies all updates done in China and Russia to her data in France.

Scenario 3 (Data center expansion): A data center in France is nearing maximum storage capacity, and so a data center in Spain is created and the system migrates some users from France to Spain. During this migration, the two above scenarios may happen with some of the users being migrated from France to Spain.

More generally, migrations can be *ephemeral* or *permanent*. Ephemeral migrations are reversed in the future; they are implemented by creating a cache of the user data at a new location and possibly pre-fetching parts of the data. Later, the cache is flushed if it has dirty data and then removed. Permanent migrations are not reversed; they are implemented by copying the user data to the new location, while coordinating updates to the data so that they go to the right location. Sometimes, a migration may start off as being ephemeral, but may end up being permanent—this could happen, for example, if a user travels to location and ends up staying there for the rest of her life. In that case, the cache gets transformed into the home of the data. Ephemeral and permanent migrations may occur simultaneously, say because the user is traveling but her home data center is being reassigned.

Migration must functionally appear as a no-op: reads and writes should functionally behave the same way whether or not migration has occurred or is in progress, except in terms of performance (a completed migration will improve performance by reducing the number of remote accesses). Moreover, migration is an optimization rather than a task required for correct operation of the system. We do not wish migration to disrupt the performance of the system by consuming large amounts of bandwidth during busy times. We thus expect migrations to occur in the background with low priority.

3 Distributed data overlays

In this section, we describe our approach to migration using distributed data overlays or simply *overlays*. Our description is targeted at a fairly general distributed storage system. In Section 4, we provide the details of overlays for a specific key-value store system called Nomad.

Overlays are an abstraction to provide online migration. Migration results in partial copies of data at two or more locations—such as cached fragments and partly-copied data—which need to be managed carefully while the system orchestrates accesses, to ensure writes are not lost and reads return valid data. For example, if data is written at the old location at the same time it is being

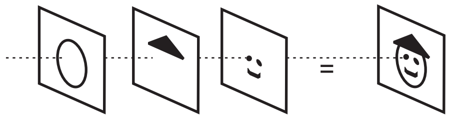


Figure 1: Overlays.

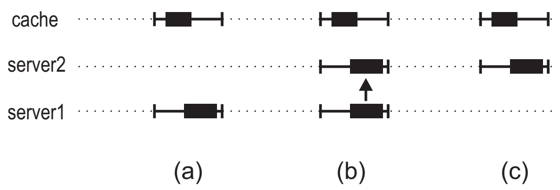


Figure 2: It is easy to use overlays to migrate data from server1 to server2. The black bars indicate regions with data. (a) Initially, data is in server1, and there is a dirty cache at a data center close to the user. (b) First, insert an overlay at server2 between server1 and the cache, and copy data from server1 to server2. (c) Then, remove the overlay at server1; dirty cache remains in place. The distributed protocols that implement overlays ensure that the insertion and removal of overlays will never cause the loss of data in ongoing read or write operations.

migrated to a new location, the system may fail to migrate the new write. This example becomes more complex when there are dirty caches, those caches themselves are being migrated, and/or migrations are canceled and restarted; the number and complexity of the different scenarios that must be handled can be problematic for the system developer. Overlays are an abstraction that helps dealing with these scenarios in a simple and unified way.

As an everyday analogy, an overlay is a sheet of transparent plastic that is placed over a piece of paper. Where it is clear, the overlay reveals the contents underneath; where it is written, the overlay overrides those contents (Figure 1). Overlays can be stacked, to create many layers, so that looking at the stack reveals their combined contents; if many overlays have content at the same place, the higher overlays occlude the lower ones.

This idea has an analogue to storage systems. We now explain it in a context where the user data is a byte sequence, such as a file, a data object, or the sequence of blocks on a disk—depending on the nature of the storage system. Data is stored at some base location and it may be partly stored in another data center, which serves as a cache. We can view the base location and cache as a stack of two overlays, as shown in Figure 2(a), where each overlay is stored in a server in a data center. For uniformity, the base layer is also called an overlay. The combination of all overlays determines what data is seen on the stack, with higher overlays having priority over lower overlays.

With the abstraction of overlays, migrating data is straightforward: (1) we create an overlay below the caching overlay, residing at the destination server, (2) we populate the new overlay by copying data from the base overlay, (3) we delete the base overlay, so that the new overlay becomes the new base (see Figures 2(b) and 2(c)).

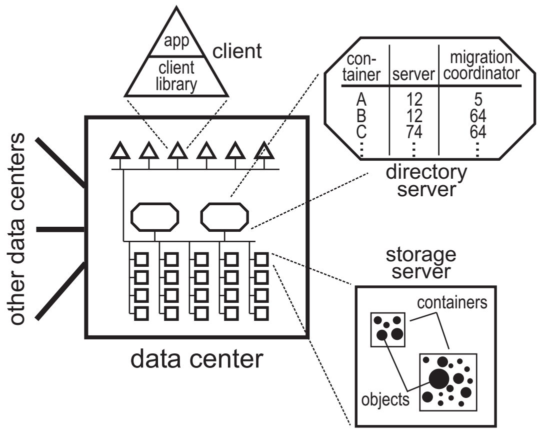


Figure 3: Standard object store architecture used in Nomad.

Function	Description
read(*container*, *oid*, *off*, *len*, *buf*)	read object
write(*container*, *oid*, *off*, *len*, *buf*)	write object
create(*container*, *oid*, *len*, *buf*)	create and write object
delete(*container*, *oid*)	delete object

Figure 4: Nomad API to access objects.

While migration occurs, data can be written at the cache, and the cache can be flushed by writing its contents to the new overlay. The protocols implementing overlays (which are hidden from the designers who just want to use them) ensure that reads and writes on an overlay stack go to the right overlay, and that overlays can be created, inserted, and removed atomically even if reads and writes occur concurrently. With overlays, it is easy to support the three scenarios described in Section 2, by inserting overlays for caches or other copies of data, and copying data between overlays to migrate. Note that each overlay is kept at a fixed server, that is, an overlay does not move; migration is achieved by creating overlays and copying data between them.

4 Nomad design

We built Nomad, a prototype of a distributed key-value object store that incorporates overlays to support flexible and online migration in a geo-distributed setting. We describe overlays in Nomad for concreteness; however, overlays are applicable to other types of storage systems, such as file systems or block storage systems.

4.1 Basic architecture

Nomad has a typical architecture for a distributed key-value object store, shown in Figure 3. This architecture is not novel; we describe it in this section for completeness.

Objects are stored on a set of storage servers, which are commodity machines running a standard operating system; they store each object as a separate file in the local file system. Throughout the paper, *client* refers to the entity that uses Nomad, which is an application running at a server in the data center, whereas *user* refers to the entity that uses the application, which is often outside the data center—for example, the user could be a person using a web mail system. Clients access Nomad via a client library that implements functions for reading, writing, creating, and deleting objects, as shown in Figure 4. There are also functions to read or write multiple objects in the same request for efficiency; these are not shown for simplicity. This interface is simple: a write associates a key with a new value, and a read returns the latest value associated with the key. Each object is part of a *container*, similar to a directory in a file system, a bucket in Amazon's S3 [5], or a blob container in Microsoft Azure [6]. A container is stored in one of the storage servers, and there is a function for enumerating the object identifiers in a container, and to create/remove containers (not shown). The mapping of containers to storage nodes is kept by a directory service, which is replicated asynchronously across data centers. For flexibility, Nomad allows any container to be mapped to any storage server; the mapping is represented as a list of container-server pairs. The client library caches part of the mapping, and the directory service need not keep track of who caches what: if the mapping changes (because the container is migrated), the client library may try to access an object at the wrong location, in which case the client library gets an error, consults the directory to find the right location, and tries again. It is possible to expire entries in the client cache for efficiency, so that the client does not use too old information, but Nomad does not do that. The directory service also indicates a *migration coordinator* for each container (Section 4.5).

4.2 Migration granularity

At what data granularity should migration occur? We discuss this issue from three perspectives: the application, the system administrator, and the storage system.

From an application perspective, applications should organize and migrate data in units that are likely to be accessed together within a given location, to provide locality. For example, in a web application, a user usually logs in at a data center close to where she lives; her persistent data, such as personal information and preferences, form a coherent unit for migration. It would not make sense to migrate a user's name without migrating her address, for example.

From the system administrator's perspective, the choice of granularity comes from a balance of manageability and control. On one hand, migration should be fine enough to allow reasonable control over the allocation of storage capacity and bandwidth. On the other hand, migration should be coarse enough so that the number of units to be administered is small.

From the storage systems perspective, the migration granularity should match the granularity of the mapping at the directory service, so that the migration engine does not have to reimplement this functionality. In particular, before migration, the directory maps some unit of data to some server; after migration, this unit must be mapped to a different server without leaving behind intermediate mappings.

Consequently, migrations in Nomad are done at the granularity of a container, and we intend that application designers collaborate with system administrators to choose an appropriate organization around such containers. For example, in some web applications, all of a user's personal information and preferences could be stored in a container. In a web mail system, there could be a container per email folder per user, so that containers are not extremely large.

A related consideration is the specificity of the destination of migration. In Nomad, the migration targets are servers, but a high-level migration decision by an administrator could be to move a container from one data center to another. In this case, there has to be a component that refines this decision and picks actual servers; this component, as well as the actual policies for migration, are orthogonal to the migration mechanisms in Nomad.

4.3 Overlays in Nomad

Our description of overlays in Section 3 assumed simplistically that the user's data and migration granularity is a sequence of bytes. We extend the description to Nomad, where the migration granularity is an object container, which consists of a set of objects, where each object is a sequence of bytes. An overlay for the container is an overlay for each object in the container plus an overlay for an array of bytes representing the set of object identifiers in the container. All objects in the container have identical overlays, except that the data contents for different objects are different. Thus, for efficiency, Nomad keeps a single *overlay structure* per container, which represents the (identical) overlays of all objects in the containers, without any data; the data is kept separately as a set of extents for each object at each overlay. An object may have several extents at a given overlay.

Overlay internal information. Recall that the directory service maps each container to a storage server, which in turn stores the base overlay for the objects in that container. Normally, the base overlay is the only overlay in the stack, but when the container is being migrated or cached, there may be additional overlays. The overlay structure consists of the following information:

- *container-id:* container that the overlay refers to;
- *location:* server that stores the data in the overlay;
- *above-pointer:* pointer to the overlay above, or nil;
- *below-pointer:* pointer to the lower overlay, or nil;
- *frozen:* a flag indicating that overlay pointers cannot be changed;

An overlay structure is associated with the following:

Function	Description
insert(*server*, *overlay*, *direction*, *flags*)	create overlay and insert
remove(*overlay*)	remove overlay
get_stack(*base_overlay*)	get entire overlay stack
start_copy(*overlay*, *direction*[, *list*])	copy to adjacent overlay
stop_copy(*copy_job*)	stop copying

Figure 5: Operations on overlays.

- *data:* a set of extents for each object, with a *unique bit* and a *timestamp* for each extent.

The unique bit is unset when the extent is a repetition of data in a lower overlay; this bit is similar to the dirty bit in a cache. The timestamp is used to handle concurrent writes when data is replicated at many overlays: the write with highest timestamp wins. We explain replication in Section 4.6.

Reading and writing data. To write data to an object, the client first finds the highest overlay O_{high}, by starting with the base location and successively traversing the *above-pointer* at each overlay until it becomes nil. Then, the client sends the data to be written to the overlay O_{high}. If the *above-pointer* at O_{high} remains nil, the storage server at O_{high} accepts the write and sets the unique bit for the newly written extent. (The checking that the above-pointer is nil and the acceptance of the write must be performed atomically with respect to the processing of other client requests for the overlay.) Otherwise, there has been a concurrent operation to insert an overlay above O_{high}, so the storage server returns an error together with the value of *above-pointer*; the client continues the traversal to find the new highest overlay, and retries the write there. When the client has completed the write, it caches the identity of the highest overlay it found. In its next write, the client starts the traversal from the cached overlay, for efficiency. The cached value could be an overlay that no longer exists (because it was removed), in which case the client gets an error and consults the directory service to find the base location again.

To read an object, the process is similar but slightly more complex, because the highest overlay may not hold the data to be read; in that case, the client goes back to the lower overlays until it finds the data it wants. It is possible that an overlay holds only part of the interval to be read, in which case the client goes to the lower overlays for the missing pieces.

Note that when there is a single overlay—which is often the case for most objects—its location is the server indicated by the directory service, and a read or write request proceeds as in a system without overlays, without additional communication rounds.

Overlay operations. The operations that insert, remove, and copy overlays are shown in Figure 5. The insert operation indicates the server for the new overlay, an existing overlay where the new overlay will be inserted, a direction (*above* or *below*) to specify whether to insert above or below the specified overlay, and a flag with properties for the new overlay. Currently, the only property is whether the new overlay holds unique data or not. If it does not, then when a write happens at the overlay, the write is also forwarded to the overlay below; this mechanism can be used to implement a write-through cache. The *start_copy* operation copies the objects in the overlay to the overlay above or below. It can copy all object or just those indicated on a list—this is useful to populate caches with certain objects only. The remove operation is self-descriptive; the system takes care of copying the overlay's unique data to the overlay below before removing it. It is not legal to remove an overlay if it is the only overlay in the stack. Not shown in the figure are the operations that return the base overlay for a container and for an object.

To simplify the design, we require that overlay operations be executed one at a time per container. This serialization occurs per container, not across containers, and so it does not pose a performance problem since overlay operations on a container are relatively rare. To serialize, overlay operations can be called by only one server per container: in Nomad, this server is indicated by the directory service and it is called the *coordinator* of the container. The coordinator ensures that an overlay has at most one outstanding overlay operation. To achieve fault tolerance, we can fail over the coordinator as we explain later. Note that read and write operations are *not* overlay operations: they can be executed concurrently with overlay operations and with each other, at many clients. The protocols that implement overlay operations, described in Section 4.5, ensure correct behavior in these cases.

4.4 Using overlays in Nomad

It is easy to use overlays to migrate data, create a cache, migrate the cache, and migrate data back, as we now describe. We provide intuitive explanations in English, but it is easy to translate these explanations into code that calls the functions in Figure 5.

Migrate data to another server. The system creates an overlay at the destination server on top of the source overlay to be migrated; at this point, writes will no longer go to the source overlay. The system then invokes the operation to copy the data from the source to the destination overlay. When the copy is finished, it removes the source overlay. As we mentioned, because we designed the overlay operations so that clients can concurrently access data, migration proceeds concurrently with these accesses, and without causing reads or writes to be lost.

Cancel migration. Sometimes, migration should be canceled because of changes in the workload. For instance, if a user is traveling for some time and migration starts, but the user returns before migration has finished, the system may decide to cancel the migration. This is easy: we simply stop the copying operation and remove the new overlay that was created for migration. Recall that the operation to remove an overlay copies the overlay's content to the overlay below. If the new overlay already has lots of data, the following optimization is effective. Note that only data written by the client needs to be copied, not data written by the migration, which is already present in the overlay below. To identify these writes, the writes by the client have a special *unique* bit set (Section 4.3), while the writes by the migration do not.

Create cache at a data center. We can create two types of caches at a data center: write-back or write-through. (Caching can also be done at the client; this can be done by the application if desired, not by Nomad.) To establish a cache, one simply inserts a new top overlay stored in the desired data center; write-back or write-through behavior is indicated by the *flag* parameter of the insert operation, which indicates whether the overlay will forward writes to the overlay below or not. To flush the cache, one invokes the copy operation to the overlay below. To remove the cache, one invokes the operation to remove the overlay.

Migrate the cache. To migrate a cache (which may have dirty data), we use the procedure to migrate the data at an overlay, described above.

4.5 Implementing the overlay operations

We now describe how to implement the overlay operations of Figure 5. We make the following design decisions: (1) it is reasonable to serialize overlay operations on the same container, but we should allow operations on different containers to run in parallel, and (2) an overlay operation on a container must allow reads and writes on the container to proceed in parallel, because these operations are sensitive to performance. Therefore, we assign a *(migration) coordinator* per container, which executes overlay operations on that container one at a time, and we design the coordinator protocol carefully so that reads and writes are never blocked. The coordinator is indicated by the directory service, and it manipulates the overlay state at each server via remote procedure calls (RPCs), as we now explain.

Inserting overlays. To insert an overlay $O2$ at storage server S above overlay $O1$ and below overlay $O3$, $O1$ and $O3$ must point to $O2$, and $O2$ must point to both. To do so, the coordinator executes the following actions (using RPCs), in this order: (1) create $O2$ at S with pointers to $O1$ and $O3$; (2) change $O1$.above-pointer to $O2$; (3) change $O3$.below-pointer to $O2$. Note that after (2) before (3), $O2$ is already visible to read and write operations because $O1$ points to it, but $O2$ is in a funny state where $O3$ does not point to it yet. This is not a problem, because $O2$ has no data and it is impossible for it to get any data (writes would go to $O3$ instead).

To insert $O2$ at the top, the process is similar except that $O2$'s top pointer is nil, and step (3) above is skipped. To insert $O2$ at the bottom, the process is also similar except that $O2$'s bottom pointer is nil, and step (2) changes the base pointer at the directory service to point to $O2$. There one subtlety: the directory service is replicated asynchronously; the coordinator changes only the directory server in its own data center and the others are eventually updated; in the meantime, the remove directory servers may temporarily point to the wrong base; this is not a problem since the directory service is used only for finding the top overlay (see "Reading and writing data" in Section 4.3).

Removing overlays (part 1). To remove an overlay $O2$, we first consider the case when $O2$ is completely occluded by the overlay above: that means all data in $O2$ is covered by data at the overlay above, so that the data in $O2$ is useless. In that case, the coordinator can remove $O2$ without fear of losing data; to do so, the coordinator (1) changes the overlay below to point to overlay above, (2) changes the overlay above to point to the overlay below and sets the unique bit for all extents in the overlay above[2]. If there is no overlay below, because $O2$ is the base overlay, the coordinator changes the base pointer at the local directory server (instead of the overlay below); the other replicas of the directory server may temporarily point to the deleted $O2$, so we leave a tombstone at $O2$ pointing to the overlay above; the tombstone is removed after a period long enough that all directory servers have seen the update (say, one hour).

Another easy case is to remove $O2$ when the overlay below is in the same storage server. In that case, the coordinator asks the storage server to execute three actions: (1) locally copy the contents of $O2$ to the overlay below $O1$, (2) redirect any writes to $O2$ so that it goes to $O1$, and (3) make $O1$.above-pointer point to $O2$.above-pointer. These three actions can be done without races because they are done in the same server. Finally, if there is an overlay above $O2$, the coordinator makes its below-pointer point to $O1$.

The removal process we described so far does not allow one to remove the top overlay, or some overlay that is not completely occluded. We come back to that soon, because such an operation uses the next operation.

Copying data between overlays. To copy data from an overlay to the overlay *below*, the coordinator asks the server of the overlay above to send the data to the server below. This idea can also be used to copy from an overlay to the overlay *above*, but it is more efficient to ask for the overlay *above* to pull the data from the overlay below, because if the overlay above already has data for certain objects, these objects need not be copied (since the overlay above occludes the overlay below at those objects).

[2] Setting the unique bit this way is a conservative choice.

Removing overlays (part 2). We can now describe how to remove an overlay $O2$ that is not completely occluded. The procedure to do that reduces to invoking existing procedures that we already described. There are two cases:

1. If there is an overlay above $O2$, the coordinator invokes the copy operation from $O2$ to that overlay, to occlude $O2$. The coordinator then uses the previously-described procedure to remove an overlay that is occluded. This works without any races, because once an overlay is occluded, it remains occluded as no writes can go to it—unless the overlay above is removed, but as we explained above, this does not happen since all operations on an overlay are serialized by the coordinator.

2. If $O2$ is the top overlay then it must have some overlay $O1$ below it at some storage server S (the last overlay cannot be removed, which would result in data loss). The coordinator first creates a new temporary overlay $O3$ over $O2$ at storage server S. Then, the coordinator removes $O2$ using the above procedure, since $O2$ is no longer the top overlay. We are left with overlays $O1$ and $O3$ at server S. The coordinator now uses the above procedure to remove an overlay when the overlay below is in the same storage server.

Copy optimization. The coordinator serializes overlay operations, but this is inefficient in one case: the copy operation can take a long time and hence delay further overlay operations. For example, suppose that we have a base overlay and a cache, and we want to migrate the base from a server to another one. During this migration, we may want to also migrate the cache to another place, but if the overlay operations on the same container are serialized, the cache migration must wait for the server migration to finish, which is undesirable. We address this problem by having copy operations run in the background, thereby allowing concurrent execution of other overlay operations on the same container. For this to work, we need to restrict the other overlay operations so that they do not change the source and destination overlays involved in a copy (for example, it would be problematic to remove the source or destination overlay while the copying is going on). We do this simply by setting a "frozen" flag at the overlay; an overlay operation that encounters the frozen flag exits with an error and retries later. It suffices to freeze the lower of the two overlays, because the operations to remove the higher or lower overlay or to insert an overlay between them will access the lower overlay first and find the frozen flag.

Correctness proof. The operations to insert, remove, and copy data between overlays ensure that reads and writes behave equivalently as if they were executing in a *single-overlay system*, that is, a system that has a single fixed overlay where all the reads and writes occur.[3] As a consequence, read and write operations are linearizable [20], which provides a strong form of consistency. Roughly speaking, linearizability ensures that each operation appears to take place instantaneously at a point between the invocation and response of the operation.

To show the property of equivalence to a single-overlay system, we examine the steps of the protocols to insert, remove, and copy data between overlays, and we show that each of these steps always cause a concurrent write or a read operation to occur at a proper overlay: a write always occurs at an overlay that is not occluded (at the time the write is applied to the overlay), so that the write behaves equivalently as in the single-overlay system; and a read always occurs at the highest overlay with data. The proof requires an exhaustive examination of all cases, which is long but conceptually simple.

Availability and fault tolerance of migration. We optimize to provide high-availability for reads and writes; migration operations may pause due to failures. A coordinator crash affects only its own migration operations: we designed the protocols so all clients continue reading and writing consistently without blocking if the coordinator crashes. We recover from coordinator crashes using standard techniques. The coordinator logs each operation and each step within the operation; the log is stored in Nomad itself. If the coordinator crashes, another coordinator reads the log and picks up from where the crashed coordinator left off.

Moving the coordinator. There is a unique coordinator per container, indicated by the directory service, but the coordinator can be easily changed, as follows. The old coordinator finishes its current operation and then performs three actions: (1) start the new coordinator, (2) change the pointer at the directory service, (3) stop.

4.6 Replication

Data replication can be implemented at two places in the component stack: at the storage node level, called *node-level replication*, or at the directory level, called *directory-level replication*.

With node-level replication, a storage node is responsible for replicating itself, and all the replicas are treated by the higher layers as a single virtual node. The migration engine is above the replication mechanism, and we migrate data from one virtual node to another as if the node were not replicated at all. For example, if there are two replicas r1 and r2 of a storage node, they are both treated as virtual node r; containers in r can be migrated to another virtual node s that could have replicas s1 and s2. The advantage of node-level replication is that it is extremely simple and modular, because migration is decoupled from replication. For example, node-level repli-

[3]This holds when overlays are not replicated. Replication is discussed in Section 4.6. It provides a consistency guarantee that is dictated by the replication scheme; for instance, asynchronous replication provides only eventual consistency.

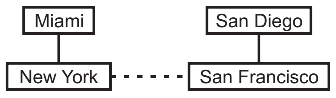


Figure 6: Replication of base overlays. The overlays in New York and San Francisco are asynchronous replicas. There are overlays in Miami and San Diego used as caches of these locations.

cation can be easily provided using a disk array at the storage nodes, or by using state machines coordinated via Paxos [24]. The drawback of node-level replication is that it cannot benefit from the versatility of overlays—for instance, we cannot use overlays to cache or migrate individual replicas, since these replicas are abstracted at a low level in the system.

With directory-level replication, the directory service maps an object/container to several servers (instead of a single server) holding replicas of the base overlay; these replicas are coordinated by the read-write protocol used by clients. The migration engine is below the replication mechanism, and migration moves data from one physical storage server to another. With directory-level replication, individual replicas can benefit from overlays, as illustrated in Figure 6. This scheme is particularly useful when data is replicated across data centers. We now explain how Nomad can be extended to support replication of this sort. (This extension is not implemented in our prototype.) The replicated base overlays are established when a client creates a container and indicates that it should be replicated. Each replicated base overlay may subsequently have a different stack of overlays on top of it, so the stacks are not copies of each other. The data in the different overlay stacks are kept in sync using the desired replication scheme. We believe overlays should work with most replication schemes, by treating each overlay stack as a black-box to which the desired replication protocol issues writes and reads. We illustrate how this is done via three well-known replication schemes—asynchronous primary-backup, asynchronous timestamped, and synchronous. Under all schemes, the directory service indicates the locations of all replicas of the base overlay; when a client writes to one of the overlay stacks, it performs the write at the other stacks as well; to write on a given stack, the client uses the write procedure described in Section 4.3. We now explain the specifics of each replication scheme.

With asynchronous primary-backup replication, one of the overlay stacks is designated as the primary and the other stacks are read-only; writes are only permitted at the primary stack, and the client applies the write asynchronously (in the background) to the other stacks.

With asynchronous timestamped replication, writes are permitted at all replica stacks, and the clients apply the writes asynchronously to the other stacks; a write includes a unique real-time timestamp to order concurrent writes by other clients at other replicas. This is a standard technique: when writes occur at different replicas, the write with higher timestamp obliterates the other writes; if a replica receives a write with a lower timestamp than the data it has, the replica ignores the write. Note that timestamps are globally unique (done by appending a machine identifier to break ties). To obtain timestamps, we assume that clocks are synchronized, say via NTP; machines with faulty clocks can be disabled using a simple monitoring service. Timestamps are kept forever for each extent. We believe that is a small overhead, but if desired it is possible to garbage collect the timestamp at a replica after it is known that the data at other replicas cannot have a smaller timestamp, using the convention that data with no timestamp is treated as having a $-\infty$ timestamp.

With both schemes above (asynchronous primary-backup or timestamped), if a client fails while writing, the write may be applied to some but not all replicas. For that reason, the migration coordinator runs a cleaner that periodically checks for these failed writes and completes them. To make it easy to recognize the failed writes, the client leaves a mark in the overlays that it writes to, which the client clears asynchronously after the client has written to all replicas. If the client crashes without having written to all replicas, the marker will be left at the overlay. Both asynchronous schemes described above provide eventual consistency.

With synchronous replication, when a client issues a write to one of the overlay stacks, the client must write to the other replica stacks synchronously (i.e., before the write is acknowledged to the client). As with asynchronous replication, a write includes a timestamp to order concurrent writes, and we use a marker to recognize failed writes. To read, a client reads from one of the overlay stacks and then checks that the data being read has no marker (the common case); if it has a marker, the client writes the data and its timestamp synchronously to the other replica stacks. This is done to ensure that later reads at other replicas cannot not return data that is older than the data being returned by the current read, to provide a strong form of consistency. This synchronous replication scheme provides linearizability [20].

With all of the replication schemes, we can migrate a replica using the procedure described in Section 4.4.

4.7 Multi-way caching and split overlays

In Section 4.4, we described how to use overlays to cache data at one location. It may be desirable to set up multi-way caches, where data is cached at many locations from a single replica. In other words, there is a single replica of the full data set, and many caches each with some (possibly overlapping) part of the data set. To do this, we need the notion of a split overlay, which is illustrated in Figure 7. (This extension is not implemented in our prototype.)

Caches exist for performance, and so they should al-

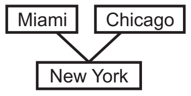


Figure 7: Split overlays are used to cache the data in New York at both San Francisco and Chicago.

low for efficient reads and writes without synchronization across the caches in different data centers. As a result, split overlays provide only a weak form of consistency, namely eventual consistency. A split overlay occurs when an overlay has several overlays over it on the next level; each of these overlays is called a *split*. A split is established using the operation to insert a new overlay of Figure 5, with a special flag indicating it is a split. This flag causes the overlay below the point of insertion to store an additional *above-pointer* to the new overlay. When a write occurs at one of the splits, the write occurs as in any other overlay: the data is marked as unique (dirty) and it is *not* propagated to other splits. A client can cause the data at a split to be copied to the common overlay below, via the *start_copy* operation of Figure 5. This corresponds to flushing the cache. When the overlay below receives the data, it invalidates older data at the same position in the other splits, using the data's timestamps to decide what is older. As a result, content initially written to a split is not visible at the other splits, but as soon as the content is flushed down, it becomes visible. In the example of Figure 7, when the dirty writes in Miami are flushed to the common overlay in New York, New York sends an invalidation message to Chicago, which causes Chicago to discard any older writes. Subsequent reads in Chicago will read the data from New York.

In general, the protocol works as follows. Suppose there is an overlay O at level k and m splits $O_1, \ldots, O_m$ at level $k+1$. If a write occurs at an overlay O_i or above, the write remains in the split with the unique bit set. Subsequently, when the data at O_i is copied to overlay O, the server of overlay O sends an invalidation message with the data's timestamp and position to the other splits at level $k{+}1$. Each of these overlays checks whether it has older data in the same position and, if it does, removes such data from the overlay. If there are further overlays above, the invalidation message is forwarded recursively. If the server of an overlay crashes and recovers, it may lose this invalidation message (e.g., it may have received the message and then crashed without having the time to process it). For that reason, the overlay that originates the message retransmits it periodically until it gets acknowledgements from the top overlays in each branch. An overlay may process the same invalidation message twice, but this is not a problem since the message is idempotent.

Now suppose that we want to remove a split overlay—say, in the example of Figure 7, we wish to remove the split in Miami and be left with an unsplit stack with New York and Chicago. We use the procedure to remove an overlay described in Section 4.5, with a small modification to incorporate the invalidation mechanism that we described above. More precisely, as shown in Figure 8, we first create an overlay in New York on top of Miami; we then copy the data from Miami to the overlay above it in New York; next we remove the overlay in Miami. We are left with a split overlay where New York is on top of New York, as shown in Figure 8(c). The final step is to locally copy the data from the higher to the lower overlay in New York, send invalidation messages to Chicago, and finally remove the higher overlay in New York.

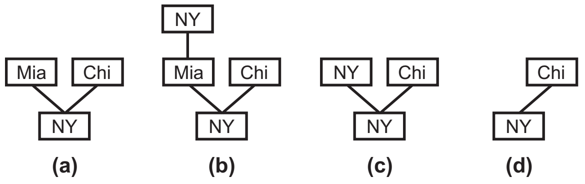


Figure 8: Removing a split overlay. Miami (Mia) and Chicago (Chi) are initially split and we wish to remove Miami.

Note that split overlays can be migrated as well, using the procedure described in Section 4.4. This modularity makes overlays a flexible mechanism.

5 The policy of geo-distribution

Thus far, we have described the Nomad system and the *mechanism* of migration it provides. We now discuss the *policy* of migration: what data to migrate, where to migrate it, and when to do so. There is no one-size-fits-all policy: migration policies depend on the specifics of data center deployments as well as application requirements. Below, we describe some of the key deployment factors that a policy layer must take into consideration.

Data center granularity: A geo-distributed system may consist of a few large data centers, or many small data centers. The former characterizes current deployments of large companies such as Microsoft, while the latter alternative is based on the use of containers [12]. Smaller data centers allow data to be closer to users, but place greater strain on the migration scheme.

Network costs: A geo-distributed system usually communicates on two different networks: an internal one between data centers, and an external one to connect with users (the Internet). The cost model for the internal network can vary. If the internal network consists of dedicated, privately owned links, the cost and speed of the network are fixed. Alternatively, network cost on leased links can depend on the amount of data transferred; for example, it is common for network operators to bill customers based on 95th percentile network utilization. The external network is provided by Internet ISPs, and the cost depends on the amount of data transferred in and out.

Access protocols: When a user accesses the service via the web, the request is redirected via DNS-based load-balancing to a local data center. If the data the user needs (e.g., her inbox) is in a different data center, the system has two options:

- *Redirect.* The system redirects the user to the appropriate data center. Subsequently, the local data center is not in the communication path, and the communication from the user to the appropriate data center is via the Internet. This option saves bandwidth on the internal network, but may impair the user experience because the Internet provides no quality of service.
- *Relay.* The local data center continues to serve the user and fetches needed data from the remote data center using the internal network. Thus, the local data center is in the communication path. This option tends to provide more predictable access times, and it allows the local data center to satisfy parts of the request locally (e.g., ads). However, this option is more expensive because one must provision the internal network adequately.

Against this backdrop, a migration policy must trade off migration bandwidth on the internal network for reduced access latencies. If the system uses the *Relay* option, the policy also has to factor in the bandwidth cost on the internal network of remote accesses on non-migrated data; in the *Redirect* model, this is not a factor.

A policy layer for online services. In addition to the deployment factors listed above, migration policies also depend on application characteristics. For example, Nomad could be used with a policy layer that periodically computes optimal placements for data given the location of recent accesses of users and the capacity of each data center, as in the Volley system [7]. The effectiveness of this policy depends on the application; it works well if user movements tend to be permanent, but can result in excessive migration if users move back and forth.

We describe a new policy layer based on predictions of future user movement. This layer provides insight into how predictions of user movement can be used to achieve access locality while eliminating unnecessary migrations.

Our policy layer makes the decision to migrate a user based on the cost of doing so versus its predicted future benefit. If we could perfectly predict the benefit, this choice would be easy; since we cannot, we must settle for heuristics that use past behavior to try to predict future accesses at the same location. We consider three simple migration policies. They all monitor the location of the user when she accesses the data, and trigger migration when a condition is met. The three conditions we consider are the following:

- *Count:* Data is accessed from the same remote location a certain number of times (e.g., 10 times);
- *Time:* Data is accessed from the same remote location for a certain period (e.g., 10 days);
- *Rate:* Data is accessed from the same remote location above a certain rate (e.g., 3 accesses per day).

For example, suppose Alice moves from Redmond to London. Suppose she accesses her mailbox twice on each of the first five days in London, twelve times on the sixth day, and then returns to Redmond on the seventh day. The Count-based policy with a threshold of 10 accesses migrates her mailbox to London on the fifth day; the Time-based policy with a threshold of 10 days does not migrate her mailbox. The Rate-based policy with a threshold rate of 3 accesses per day migrates her mailbox to London on the sixth day. In this case, the Time-based policy is the best. Since Alice returns to Redmond after a short trip, her mailbox should not be migrated. In other cases, the Count and Rate-based policies may work better.

We later report on the efficacy of these different policies when applied to real user traces taken from a large web mail service. Since these policies are predicated on the movement of users in the real world (rather than the semantics of a specific application like webmail), we believe the results to be relevant for other web applications, such as the ones mentioned in Section 2.

6 Implementation

We implemented overlays in a prototype of Nomad as we described in Section 4, except that we did not implement replication (Section 4.6) and split overlays (Section 4.7)—which are unnecessary to compare Nomad to other migration schemes. The Nomad prototype has 6,000 lines of C# code, comprising a client library, a storage server, and directory server. The directory server provides RPCs to get and set the location of the base overlay of a container given its 64-bit identifier. A storage server provides RPCs for the following: (1) Read/write to an overlay; (2) Get the overlay above and below of an overlay; (3) Delete an overlay; (4) Create new top overlay at another storage server for a given overlay; (5) Copy data of an overlay to its upper overlay; (6) Migrate an overlay to another storage server.

Storage servers store data for an overlay as a directory in the local file system, containing a metadata file and one file for each extent of the overlay, named by the object id, start offset and end offset. A write to an overlay may merge extent files. Storage servers cache overlay metadata in memory to improve read performance.

7 Evaluation of mechanism

In this section, we evaluate the use of overlays for migration, through experiments that measure overlay overheads, verify their flexibility, and compare their performance against alternatives.

7.1 Alternative schemes for migration

We consider two alternative schemes for migration, which are often used in practice:

- *Lock-based migration:* While the data is copied from the old location to the new location, the system blocks write operations. Read operations are not blocked; they are served at the old location. Writes are unblocked after the old location is marked as invalid and the directory is updated to point to the new location.
- *Log-based migration:* The system creates a log at the old location to store the updates while the data is copied from the old to the new location. During the copying, reads and writes are served at the old location. Once the copying is finished, the system blocks write operations, copies the log from the old to the new location, marks the old location as invalid, modifies the directory to point to the new location, and then unblocks write operations.

7.2 Experimental setup

Our setup consists of machines in data centers in five locations: Mountain View (CA), Redmond (WA), Boston (MA), Cambridge (UK), and Beijing (CN). Each machine consists of a PC with two quad-core 2.27 GHz Xeon processors, 16 GB of RAM, an internal disk array with several 10,000 rpm SAS disks, running 64-bit Windows Server 2008 R2. Machines are connected to a Gigabit switch, and the various locations are connected by a dedicated network. The median ping latencies between locations are as follows, in ms:

	WA	MA	CN	UK
CA	19	112	167	237
WA		79	141	204
MA			220	283
CN				345

7.3 Overhead of overlays

We now evaluate the overhead imposed by overlays.

Access latency. In this experiment, we measure the latency that overlays incur on accesses to data. A client reads or writes a small object with up to five overlays in different locations, as we measure the latency of reads or writes in two separate experiments. The client is in the same location as the top overlay, which is typical of having a cache at the local data center.

Figure 9 shows the results for writes. We see that the first write incurs a higher latency, because the client needs to traverse overlays in different locations from bottom to top. Once the client learns the top overlay, it caches it for the entire container; subsequent writes on any object of the container are much faster, incurring only a local-area-network latency plus a disk-write latency. We can avoid the higher latency for the first write by keeping a hint of the location of the highest overlay at the directory service. We implemented this optimization, but Figure 9 shows the unoptimized scheme, representative of the worst case.

For reads, the situation is similar (not shown). The first read discovers the overlays, while subsequent reads are

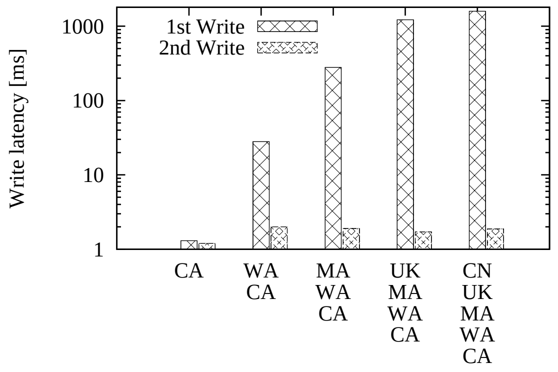


Figure 9: Write latency using several overlays in many geographical locations. The x-axis indicates the number and location of overlays.

faster. The exact latency for subsequent reads depends on the first overlay that has the data: if it is the top overlay, it is just a local-area-network latency, otherwise it is the sum of the latencies to communicate with each successive overlay until data is found. We implemented an optimization so that if a read does not hit the top overlay then it writes the data there so that the next read of the same data will be faster—thereby treating the top overlay as a cache. A client-settable flag determines whether this optimization is enabled or not.

Overlay space. We measure the space overhead of overlays. The on-disk or in-memory metadata for each overlay is smaller than 1 KB. A larger overhead occurs because lower overlays may store useless data occluded by higher overlays. In theory, a container's storage space across all servers could be multiplied by the number of overlays. In practice, most overlays are usually empty, but even if they were not, it is easy to introduce a garbage collection mechanism that periodically detects and erases occluded data (we have not implemented this). The garbage collection period can be many minutes because most storage systems are over-provisioned.

7.4 Flexibility of migration mechanism

In terms of functionality, overlays provide the flexibility to migrate data while clients are concurrently reading and write data; during migration, the system may create or flush a remote cache, and the cache itself could be independently migrated. Lock-based and log-based migration do not provide this flexibility, but they could support a static cache layer, which cannot be removed or added. With extensions, lock-based and log-based migration could support migration of the cache layer, possibly concurrently with migration of the storage layer, but this requires additional careful design. The flexibility of each scheme is summarized in the table below.

Scheme	support static cache	create cache layer	remove cache layer	migrate cache
Lock-based	Yes	No	No	Extension
Log-based	Yes	No	No	Extension
Overlay	Yes	Yes	Yes	Yes

We devised an experiment to demonstrate the flexibility of overlays as a user moves between four locations. The user is initially at the United Kingdom (UK), where she has 50 MB of data. Her workload consists of reading or writing small objects within some working set of size 2 MB. At time 60s, she moves to Boston (MA). At that time, we start a process to migrate her data to MA, starting with her active set, using 300 Kbps of bandwidth. This could be the maximum bandwidth a given user is allowed to consume in a system with many users. The entire migration will take around 1600s, but her active set can be copied in 60s. At time 180s she moves to Redmond (WA), but the migration UK-MA has not finished yet, so we create a cache in the WA data center and start populating it with her working set, copying the data from the MA data center at a rate of 400 Kbps. At time 300s, she moves to California (CA), and we migrate her cache from WA to CA at a rate of 400 Kbps. There are separate experiments for reads and writes. Note that in this experiment, we compress travel time so that we can fit the scenario in one small graph. In a more realistic setting, the user may remain at a location for several days, and the corresponding graph would look like the one we give, except it would have large segments depicting no interesting information while the user remains at a location.

Figure 10 shows the latency of the user's reads and writes during this scenario. The latency refers to a client running on the user's behalf at the data center closest to the user—this client could be a web application that reads and writes within the data center. We can see that initially the latency is close to 0, reflecting a local access. At

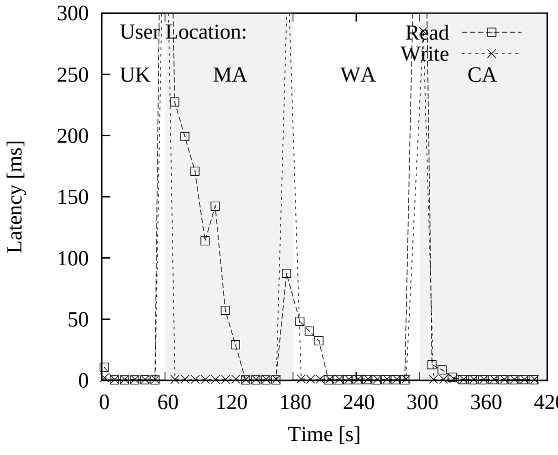


Figure 10: Read and write latency as the user moves between 4 locations and we migrate her data and cached data as she moves.

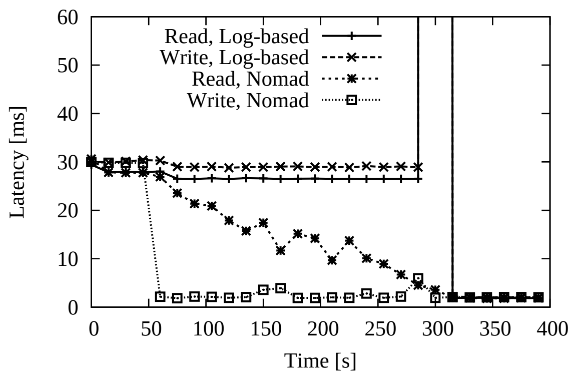


Figure 11: Read and write latency before, during, and after migration. Migration starts at 50s and completes at around 300s.

60s, once the user moves and migration starts, the latency spikes to around 1s for a few accesses: this is the time it takes to lookup the user's data and traverse the overlay stack. Soon after, the write latency drops to $\approx$0, because writes are now done locally. The read latency gradually drops as the working set is migrated from UK to MA, which takes $\approx$70s. At 180s and 300s, we observe the same phenomenon, except that the working set is copied faster, in $\approx$40s, since more bandwidth is available for populating or migrating the client's cache.

7.5 Performance comparison

In this experiment, we evaluate the latency of accesses to data during migration. A container with 50 MB of data is initially located in the WA data center (source location) and a client periodically reads or writes small objects in that container from the CA data center. At 50s, the system starts migrating the container to CA (destination location), which is the same data center as the client, using 2 Mbps of bandwidth. We measure the latency of reads and writes to the objects as the migration progresses.

Figure 11 shows the results for Nomad and log-based migration. With the latter, there is a period of write unavailability at the end of migration when the log is copied. The write unavailability is given by the formula:

$$\frac{\textit{filesize}}{\textit{migrate-rate}} \times \frac{\textit{write-rate}}{\textit{migrate-rate}} = \frac{\textit{filesize} \times \textit{write-rate}}{\textit{migrate-rate}^2}$$

where *write-rate* refers to the new writes during migration, and *migrate-rate* is the rate at which data is copied.

The unavailability can be reduced by using a second log to store updates while the log is being migrated (and this can be done repeatedly).

Log-based migration has two other drawbacks compared to Nomad. First, read and write latency remains high during migration because operations are served at the source location until migration is completed. In contrast, with Nomad, the write latency immediately decreases when the migration starts while the read latency progressively decreases, because the client reads ran-

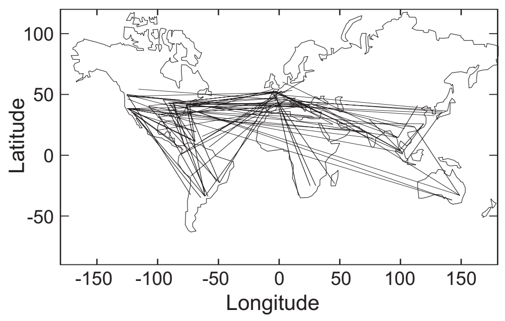


Figure 12: Movement patterns of sample users that traveled $>$2000 miles for $>$3 weeks with $>$7 accesses. Disclaimer: sample is not statistically significant, provided for illustrative purposes only, not necessarily representative of Hotmail's market penetration.

domly chosen objects and, as time passes, a larger fraction of these objects are in the same location as the client. Another drawback of log-based migration is that it consumes three times as much bandwidth for new writes done during migration (not shown on the graph). These writes must be (1) received at the source, (2) sent from the source, and (3) received at the destination. Intuitively, this is because writes during migration are sent to the source. This must be so, because reads are served at the source. In contrast, with overlay-based migration, writes can go directly to the destination, because the system can use overlays to serve reads from a combination of the source and the destination.

We also tried lock-based migration (not shown on the graph). The result is what one would expect: during migration there are no writes, and reads have high latency since they are served at the source location.

8 Evaluation of policy

We evaluate three simple migration policies using real traces from Hotmail. Generally, migration can be triggered by a combination for factors, including balancing of storage capacity, balancing of bandwidth, and movement of users. The policies we consider here are based on movement of users; a more comprehensive set of policies may consider the other factors as well [7]. We evaluate policy independent of the mechanism used for migration, to separate concerns. Our traces comprise the login records of $\approx$50,000 randomly chosen Hotmail users, collected over two months (Aug-Sept 2009). For each user, it contains the login time and IP address from which the user logged in. We use a public IP-based geolocation service to map each IP address to latitude, longitude coordinates. To apply our policies, we view each login as a separate access, and the unit of migration is a mailbox. Figure 12 shows examples of the movement patterns in the trace.

To eliminate errors introduced by the geo-location service, we first pre-process the trace by clustering sequences of close-by accesses by a user (less than 150 miles from each other) into *visits*. Thus, if a user logged in twice from New York City and twice from New Jersey (which are very close), we consider that as a single visit of four accesses. If the user then logs in from Seattle, and later again from New York City, that is three visits.

As we explained in Section 5, data center granularity is an important consideration: the movement of a user is only relevant for migration if the closest data center to the user changes. We consider that the data center changes only if the distance between one visit and the next is above a threshold. We consider three such thresholds, corresponding to three data center granularities:

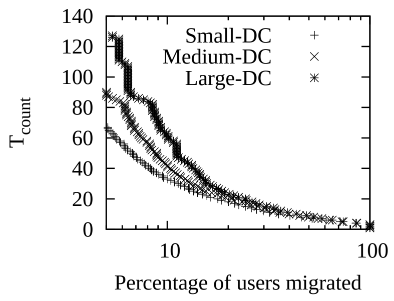
- *Large-DC:* Threshold is *2000* miles, corresponding to a deployment with massive data centers serving a large area. 1% of the users in the trace have visits that satisfy this criteria.
- *Medium-DC:* Threshold is *1000* miles, corresponding to data centers serving a mid-sized geographical region. 1.8% of the users in the trace have visits that satisfy this.
- *Small-DC:* Threshold is *450* miles, corresponding to having data centers for individual states or metropolitan areas. 3.5% of the users in the trace have visits that satisfy this.

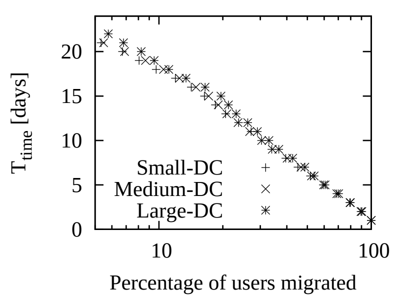
For each data center granularity, we study the three migration policies described in Section 5. For each user, we scan the trace until we find a *remote* visit—a visit whose distance from the first visit exceeds the distance threshold (2000, 1000 or 450). We then apply the policy to that remote visit to see if migration is triggered; for example, the Count policy with a threshold of 10 triggers migration if the visit contains 10 or more accesses.

Figure 13 shows what fraction of users trigger migration as a function of each policy's threshold. The fraction is relative to the users with at least one remote visit.

We now examine the effectiveness of the three policies using the metric of *saved remote accesses*, which measures the benefit of migration: these are accesses that, without migration, would have been served at the original data server far from the user, but with migration, are served from a data center close to the user. For example, if a user accesses her mailbox 500 times during a trip, and we use the Count policy with a threshold of 50, the number of saved accesses is 450.

Figure 14 shows the average number of saved accesses per migrated mailbox on the y-axis. Each point corresponds to a different threshold for each policy, for the Large-DC and Small-DC granularity (the Medium-DC is between those two, and not shown). The x-axis has the percentage of migrated users using that threshold; points to the right correspond to thresholds that migrate more users. For the Count and Rate policies, the curves decrease monotonically as more users are migrated; therefore, most of the migration benefit is obtained by choos-





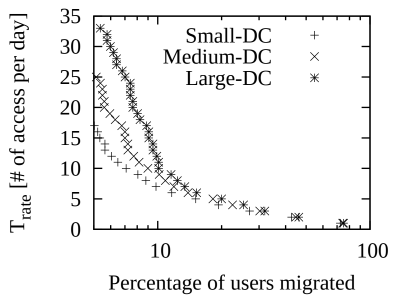


Figure 13: Effect of varying thresholds for Count (Left), Time (Middle) and Rate (Right) policies.

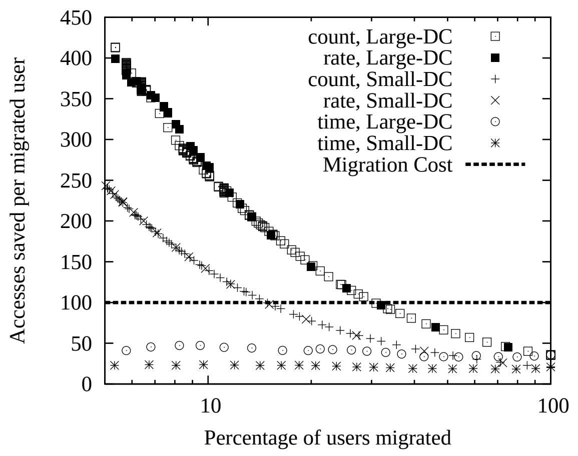


Figure 14: The benefit per migration for different policies.

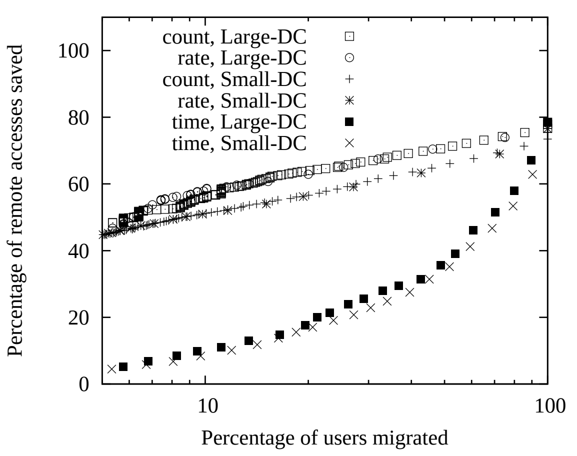


Figure 16: Effectiveness of policies in saving remote accesses.

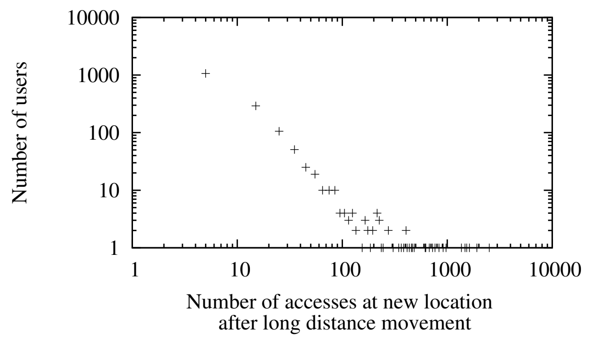


Figure 15: Distribution of # of accesses during remote visits.

ing a threshold that migrates few users, with more aggressive thresholds providing diminishing returns. The graph shows that the Count and Rate-based policies are better than the Time-based policy.

Figure 14 can be used to determine the break-even point for each policy, where the benefit of migration (saved remote accesses) outweighs the cost of migration, assuming the *Relay* access model (Section 5), in which remote accesses and migration consume bandwidth on the same internal network and can be quantitatively compared. The line marked "Migration Cost" illustrates a specific case where the bandwidth needed for migration is equivalent to 100 remote accesses (e.g., for migrating recent emails). This number could be different, corresponding to different horizontal lines. The break-even point is where the horizontal line intersects the curve of each policy.

In both the *Relay* and the *Redirect* model, the internal network may have limited bandwidth for migration if other traffic is given priority. In this case, migration can be done opportunistically, using spare bandwidth during intermittent idleness of the links. Which mailboxes should be migrated to offer the greatest benefit? Figure 14 indicates that mailboxes migrated by the largest thresholds offer more benefit than those migrated with smaller thresholds. This argues for adaptively changing the threshold to match available bandwidth.

Figure 15 explains why the Count policy works well. It plots the distribution of the number of accesses per remote visit; we see that the distribution is linear on a log-log scale and can be fitted to a heavy-tailed Pareto distribution, with a few visits containing many accesses. This explains the monotonically decreasing benefit of the Count policy on the previous graph: it can be analytically shown that a Pareto distribution always exhibits this property (we omit the analysis for lack of space).

Finally, we determine the overall effectiveness of the different policies by measuring the total percentage of remote accesses saved. Figure 16 plots this metric on the y-axis. The Count and Rate policies are very effective in saving remote accesses; for example, with thresholds that migrate 10% of all users, both policies save 55 to 60% of all remote accesses in the Large-DC case. As expected, the Time policy is not very effective, requiring almost all users to be migrated to achieve similar savings.

9 Related work

Migration mechanism. There has been a lot of work on distributed file systems [15, 17, 18, 21, 23, 28, 32–34]. These systems either do not support migration, or employ lock-based or log-based migration. For example, AFS [21] allows a volume to be moved from one server to another using log-based migration. xFS [33] allows a client to borrow a file for exclusive writing, but this is different from migration since the file is ultimately returned to its home server, which serves as a coordination point (e.g., if multiple clients want to write). In Pangea [32], migration is achieved by simply creating a new replica, but the system provides only eventual consistency, in contrast to Nomad. Ceph [34] allows (the metadata of) a directory to be moved from one server to another, using lock-based migration. Coda [28] allows clients to hoard files for disconnected operation; this is different from migration since hoarded files are eventually returned to the server that owns the file. Farsite [17] appears to support migration of metadata, by changing the mapping from identifier prefixes to servers, using a lock to avoid races. GFS [18] appears to support migration of chunks, by copying a chunk from one server to another, and then updating the mapping from chunk id's to servers at the master using a lock to avoid races.

Migration of a virtual machine (VM) is a well understood technology, done by VMware [3], and a couple of years later in Xen [13]. This technology is about moving a functional VM to another host. In a first round, the entire VM's memory is copied; if a page of memory changes after being copied, it is marked dirty and the marked pages are copied in a subsequent round. The system may execute many rounds as further pages are marked, until it decides to pause the VM, copy the remaining dirty pages, and start the destination VM. Subsequent work on VM migration considered the copying of direct attached storage [22]. This body of work is different from ours because it focuses on migration of data accessed by a single machine whether in memory or disk, whereas we consider a *distributed* setting and must address the required coordination among several servers (which we do via overlays).

In PNUTS [14], data is replicated across data centers and migration consists of changing the master replica. This scheme requires many replicas across data centers, which we must avoid. Cloud storage services support migration between different locations. In Amazon's storage server [4], the approach to migrate an elastic block store (EBS) is as follows: (1) the user stores a snapshot of the EBS in S3, and (2) the user creates an EBS at a different location and populates it with the content in S3. This scheme, though simple, will fail to migrate any writes done on the original EBS during migration.

Distributed object systems support migration of objects (see [11, Section 5.2.2]), which is more complex than migrating data, since objects have threads, TCP connections, and other contextual state. The migration mechanism employed is lock-based migration.

Commercial disk array solutions such as the HP-UX logical volume manager [26] support online migration by essentially using the logging technique. In this context, Aqueduct [25] is a system that controls migration traffic to maintain low access latencies during migration.

The work in [8, 19, 27, 30] shows how to add or remove replicas in a replicated state machine or a quorum system. These techniques can be used for migration, by adding a replica at a new location and removing from the old location. This work is theoretical and would be inefficient for wide-area-network storage.

Migration policy. Volley [7] uses system logs of accesses to determine placement of data across data centers, based on data access interdependencies, who has accessed the data and when, and a balance of storage capacity across data centers. This is different from our work because of four reasons: (1) Volley's placement algorithm computes a global placement for *all* data, whereas our scheme determines where a particular piece of data should be migrated, (2) Volley's algorithm does not consider the cost of migrating data, so the algorithm is not applicable to our setting where migration *has* as a cost; in fact, the consideration of cost-benefit of migration is central to our scheme, (3) Volley does not propose mechanisms for migration, (4) Volley does not attempt to predict future user movement.

Previously, data placement has been extensively studied in the context of web servers and Content Delivery Networks (CDNs) [29]. Since data in these settings is read-only, most of these solutions are centered on replica creation and placement.

Predicting the movement of users has been explored in mobile systems [10]. In contrast to this work, we are concerned with predicting movement at coarse grain (e.g., is user staying in Asia or returning to Europe?) instead of precise locations.

10 Conclusion

This paper addresses the problem of providing online migration of data across data centers—a problem that occurs as users move and/or data centers become unbalanced due to new applications, unforeseen growth, and new data centers. To design a migratable storage system, we propose an abstraction called distributed data overlays, which has a simple real-world analogy based on transparent pieces of paper. We implemented this abstraction within a prototype of a key-value object store called Nomad, which spans multiple data centers and allows for migration and caching of object containers across data centers. It is very easy to use overlays to implement migration; the complexity is hidden by the protocols that implement overlays (which we provide), as these protocols must coordinate concurrent reads, writes, migrations, and the dynamic creation and removal of remote caches. We also study some policies that might trigger the migration mechanism based on user movement, but other policies could be applied as well [7].

Acknowledgements. We are grateful to Asim Kadav, Jean-Philippe Martin, Amar Phanishayee, and Fang Yu for helpful comments on an earlier draft. We are also grateful to our shepherd Wilson Hsieh for having provided many useful comments throughout the paper.

References

[1] `http://www.verizonbusiness.com/terms/us/products/internet/leasedline/`.

[2] `http://www.swisscom.ch/solutions/Resources-en/Dokumente/factsheet/00276-factsheet-private-line-international-en.`

[3] VMware news release, VirtualCenter, Nov. 2003. `http://www.vmware.com/company/news/releases/virtualcenter.html`.

[4] Amazon elastic block store, Jan. 2011. `http://aws.amazon.com/ebs`.

[5] Amazon simple storage service, Jan. 2011. `http://aws.amazon.com/s3`.

[6] Microsoft azure blog storage, Jan. 2011. `http://msdn.microsoft.com/en-us/library/dd135733.aspx`.

[7] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Symposium on Networked Systems Design and Implementation*, pages 17–32, Apr. 2010.

[8] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, 58(2), Apr. 2011.

[9] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems*, 27(3):5:1–5:48, Nov. 2009.

[10] D. Ashbrook and T. Starner. Using GPS to learn significant locations and predict movement across multiple users. *Personal and Ubiquitous Computing*, 7(5):275–286, Oct. 2003.

[11] R. S. Chin and S. T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, Mar. 1991.

[12] K. Church, A. Greenberg, and J. Hamilton. On delivering embarrassingly distributed cloud services. In *ACM Hot Topics in Networks Workshop*, pages 55–60, Oct. 2008.

[13] C. Clark et al. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation*, pages 273–286, May 2005.

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *International Conference on Very Large Data Bases*, pages 1277–1288, Aug. 2008.

[15] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *ACM Symposium on Operating Systems Principles*, pages 202–215, Oct. 2001.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, Oct. 2007.

[17] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 321–334, Nov. 2006.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29–43, Oct. 2003.

[19] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, Dec. 2010.

[20] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[21] J. H. Howard, M. L. Kazar, S. G. Menees, A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[22] A. Kadav and M. M. Swift. Live migration of direct-access devices. *ACM SIGOPS Operating Systems Review*, 43(3):95–104, July 2009.

[23] M. Kim, L. P. Cox, and B. D. Noble. Safety, visibility, and performance in a wide-area file system. In *USENIX Conference on File and Storage Technologies*, pages 131–144, Jan. 2002.

[24] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[25] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: online data migration with performance guarantees. In *USENIX Conference on File and Storage Technologies*, pages 219–230, Jan. 2002.

[26] T. Madell. *Disk and file management tasks on HP-UX*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1997.

[27] J.-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *International Conference on Dependable Systems and Networks*, pages 325–334, June 2004.

[28] L. B. Mummert, M. R. Eblig, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *ACM Symposium on Operating Systems Principles*, pages 143–155, Dec. 1995.

[29] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *IEEE International Conference on Computer Communications*, pages 1587–1596, Apr. 2001.

[30] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, Dec. 2003.

[31] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *International conference on Architectural support for programming languages and operating systems*, pages 48–58, Oct. 2004.

[32] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mhalingam. Taming aggressive replication in the Pangaea wide-area file system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 15–30, Dec. 2002.

[33] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, pages 71–78, Oct. 1993.

[34] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, Nov. 2006.