BaseFS - Basically Available, Soft State, Eventually Consistent Filesystem for Cloud Management

Marc Aymerich Gubern
Universitat Politecnica de Catalunya UPC
marc.aymerich@est.fib.upc.edu

ABSTRACT

BaseFS is a peer-to-peer distributed filesystem for cloud configuration, designed to operate under the network conditions and administrative requirements commonly found on Wireless Community Networks. Nodes do not need to trust each other, the core data-structure is an append-only specialized Merkle tree with monotonic and cryptographic properties that allows for efficient and secure verification of data sent by untrusted nodes. Decentralized write permission is achieve using a hierarchy-based public key infrastructure built into the Merkle tree, allowing for automatic resolution of write conflicts based on *proof-of-authority*. Finally, a gossip layer provides scalable change dissemination and group membership, with time and load constant relative to group size. With no single point-of-failure, BaseFS can provide levels of availability and scalability never seen before on a cloud configuration tool.

1. INTRODUCTION

One of the steps towards building a successful distributed system is establishing effective configuration management that allows the system to scale and evolve while maintaining operations costs low. It is a complex engineering process responsible for planning, identifying, tracking and verifying changes in the software configuration, as well as maintaining its integrity throughout the life cycle of the system [25].

Some successful tools exist to aid in this process, e.g. Chef and Puppet only to name a few [9]. In these solutions, the system configuration is written in recipes that converge every few minutes. While this approach works well for static configuration, it fails to provide an ideal solution for more dynamic state, where a near real-time convergence is desirable. Because of the need for faster provisioning, elasticity in cloud environments or quickly respond to failures, systems like Zookeeper, etcd or Consul, that target this very specific problem, have emerged [15]. They are distributed key-value stores designed for keeping the global state of the system. We can make a rough distinction between the *static configuration management* tasks solved by tools like Chef or Puppet and the *dynamic configuration management* commonly solved by key-value stores like Zookeeper, etcd or Consul.

Existing *dynamic configuration management* solutions are designed with strong consistency models and client-server architectures. They have server nodes that require a quorum of nodes to operate (usually a simple majority). They choose consistency over availability under the face of a network partition. Design decision based on the assumption that these systems are deployed on a data center-like environment, where machines are homogeneous, with predictable performance, connected by fast networks, with low churn and operated by a team of highly skilled engineers, while all being part of a single administrative domain. But these assumptions are not always true.

Community cloud computing[16] is an emerging model where infrastructure is built using a collaborative effort. It is often the result of individual users providing spare resources to a common pool. As we can imagine the set of constrains faced by this kind of distributed systems are different from those we can find in the typical data center. Hardware is heterogeneous, it tends to be consumer-grade with higher failure rates and lower performance. Resources range in quantity and quality from one node to another. Nodes enter and leave the system more often. The network might be slow and unreliable; partitions may occur more frequently. The administrative boundary between organizations is sometimes blurry, with requirements for a decentralized administration of the infrastructure. Limitations on the technical capacity for effectively deploying and managing complex distributed systems may also exist, since the operators are sometimes members of the community that volunteer their time, but with limited SLA commitment. In short, community cloud architecture is peer-to-peer[3, 19], in contrast to the centralized model of traditional clouds.

The main contribution of this thesis is to provide a novel approach to solve cloud configuration management problems on a decentralized, more networked constrain, environments. First we present a case for a more available and less consistent configuration management solution. Next, we introduce the design an implementation of BaseFS, an eventually consistent gossip-style distributed filesystem specifically designed for cloud and configuration management. In section 4 we show how BaseFS can be used to manage a cloud service. Experimental results from a prototype implementation are presented in section 5 and finally we reflect on the future of BaseFS.

2. BACKGROUND

Zookeeper, etcd and Consul are consolidated distributed key-value stores for shared configuration and service discovery. But they present limitations in the context of community cloud. The more relevant, and the ones we hope to address, are: a) geographical and administrative scalability, b) trading consistency over availability and c) deployment complexities.

2.1 Scalability Limits

Existing work rely on fault-tolerant, distributed coordination algorithms like Paxos[12] and Raft[18] are used because of their strong consistency properties. But coordinated consensus is expensive, processes can't make progress independently: a majority of nodes have to agree on every decision first. Constant communication between nodes is needed, making the system hard to scale beyond small clusters or across wide-area networks. Coordination algorithms are notoriously hard to implement[18], and even harder to make them tolerate Byzantine failures. In the end, nodes need to trust each other, making it hard to scale as the number of administrative domains increases. The real scalability challenges faced by community cloud computing are not about the size of the system, but on **geographic** and **administrative** scalability.

By removing coordinated consensus, geographic scalability improves naturally, as progress is no longer restricted by network delay anymore. On the other hand, administrative scalability can be improved by removing the need for nodes having to trust each other.

2.2 Availability Under Network Partition

The CAP theorem is a valid and useful tool for reasoning about fundamental trade-offs made on the design of a distributed system[8], although it has recently been the subject of scrutiny and debate regarding whether it is overstated or not[11]. The acronym stands for:

- Consistency: all nodes see the same data at the same time
- Availability: node failures do not prevent survivors from continuing
- Partition tolerance: the system continues to operate despite message loss due to network failure

The theorem states that a distributed system facing a network partition has to choose between staying available or being consistent. In our case all the current solutions err on the side of consistency. These solutions are commonly called CP (Consistent but not available under Partition). The main implication is that in case of partition nodes under a minority partition will not be able to make progress.

CP systems are a fragile and complex piece of the infrastructure, and making a system depend on them makes progress impossible for minority partitions. It is important to stress that consistency presented by the CAP theorem actually refers to **strong consistency**. This consistency definition can be relaxed and allow availability and some kind of consistency less than "all nodes see the same data at the same time". A typical example is eventual consistency, which guarantees that after some undefined amount of time all replicas will converge on the same value.

Cheap wireless links is the network infrastructure of choice for some community cloud deployments. Nodes continuously entering and leaving the system are also expected. With unstable quorums, latency, packet loss, low bandwidth and network partitions a CP system deployed on these conditions will have a hard time staying available and deliver good performance. In this situation a cloud management solution that **focuses on availability** while at the same time provides a low conflict rate, fast convergence, and low divergence time will be desirable.

2.3 Complexity

Existing configuration management solutions are complex to deploy and maintain. They need dedicated quorum servers that have to be protected from untrusted parties. Extra efforts need to be placed on making sure network partitions do not occur, the entire system's availability may depend on it. The use of non-standardized APIs that operators need to learn also increases its complexity. Networked APIs such as REST or RPC don't come for free, applications need to account for network error conditions and optimize for IO overhead.

Additionally, because these tools are designed with data center conditions in mind, they need to be secured and tuned to operate across wide area networks.

While all this complexity has not been a problem for corporations with in-house teams of well paid, highly skilled, engineers, Community cloud is sometimes build and operated by volunteers, and there is not always good incentives for investing large amounts of effort into solving complex technical problems.

Complexity can be lowered by removing the need for dedicated servers and make the system P2P. The lack of a single point-of-failure and nodes not having to trust each other, are precisely some of the main attributes that led Bittorrent to achieve massive adoption. On the other hand, a filesystem API is something developers are already familiar with, and all programming languages have libraries for. Several projects exist that satisfy this desire on existing solutions, zkfuse [1] for ZooKeeper, etcd-fs [13] for etcd and consulfs [24] for Consul.

2.4 Existing P2P Filesystems

Before reinventing the wheel with a new solution, we examine if existing P2P filesystems can be used for effective cloud management.

Syncthing[7] and other P2P-based Dropbox-like applications are discarded because trust between nodes is assumed by means of a shared secret. Additionally, Syncthing dissemination model is based on periodic state synchronization, a bad model for fast dissemination of highly dynamic content.

IPFS, short for InterPlanetary File System, is a peer-to-peer hypermedia protocol, addressed by content and identities[6]. At the time of this writing IPFS lacks update notification, applications have to actively fetch updates for content they are interested in. A polling model for data that changes frequently is not scalable. Another issue is the *single-point of contention* of its Merkle DAG (Direct Acyclic Graph) design. IPFS uses a Merkle DAG inspired on GIT [5], changes are linked by the *commit tree*, effectively creating a *single-point-of-conflict* for the whole filesystem. Simultaneous changes on **different files** cause conflict (branches), seriously limiting concurrent writes scalability. For a version control system having all related changes linked together by a commit tree is desirable, but for an application that allows concurrent writes from multiple nodes a *per-file point-of-conflict* is more desirable.

3. DESIGN AND IMPLEMENTATION

The main design goals are to provide a distributed hierarchical datastore that supports write permissions without nodes having to trust each other. Nodes must be able to make progress even under the face of network partitions and

the system should automatically handle conflicts from concurrent writes with at least eventual consistency guarantees.

BaseFS builds on top of ideas and concepts coming from existing technologies used by successful distributed systems that have been developed over the last decade or so. The inspiration from BaseFS comes from Bitcoin, Serf, IPFS and Consul, just to name a few. In this section we present the main design aspects of our prototype implementation, including:

1. **Log** - a specialized Merkle tree of content-addressed immutable objects. Described in 3.1
2. **View** - provides a conflict free composition of the log entries. Described in 3.2
3. **Network** - maintains membership, manages connections to other peers, uses various underlying network protocols. Described in 3.3
4. **Filesystem** - emulates filesystem operations on *view* operations. Described in 3.4.
5. **Modules Overview** - how everything is glued together. Described in 3.5

3.1 Log

To make progress independently, each BaseFS node needs to maintain a local replica of the whole filesystem. BaseFS *log* implements this data-layer which is composed by two types of hash addressable objects:

- Log entries - Nodes of a specialized Merkle tree containing the whole history of log operations
- Log blocks - File content chunks

Log entries.

BaseFS log entries contain all the filesystem metadata (or i-nodes) organized as an add-only monotonic specialized Merkle tree, with the convenience of also being CvRDT *Convergent Replicated Data Type*[21].

Our specialized Merkle tree, or hash tree, is a tree where objects are linked to each other by their hash. In contrast to traditional Merkle trees, data resides on any node of the tree, no only leaf nodes. As illustrated in figure 1, log entries (representing files, directories, links...) are linked to their parent directory, conforming to the hierarchy of the filesystem. Using cryptographic hashes has many useful properties:

- Content addressing: all content is uniquely identified by its SHA-224 hash checksum.
- Tamper resistance: all content is verified with its hash.
- Deduplication: all objects that hold the exact same content are equal, and only stored once.
- Casual ordering: the object linked is older than the object itself, hashes can not be calculated in advance.

Log entries also satisfy the definition of *Convergent Replicated Data Type*. A semilattice, partially ordered set that has a join with a *least upper bound*, with sufficient conditions:

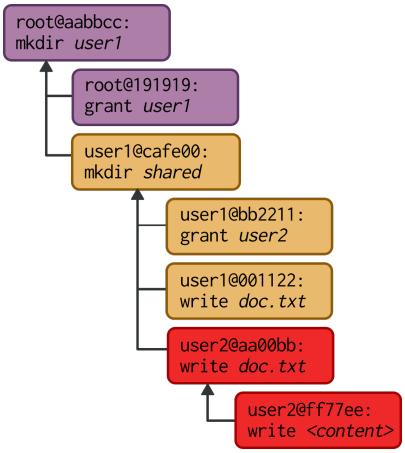

Figure 1: Partial log representation

- a) Associativity $f(f(a,b),c) = f(a,f(b,c))$
- b) Commutativity $f(a,b) = f(b,a)$
- c) Idempotency $f(f(a)) = f(a)$

Conditions that allow for a gossip-style weak communication channel with message loss, out of order, or multiple delivery. The only required condition is eventual delivery. If two nodes see the same events, they are on the same state. Characteristic known as strong eventual consistency (SEC). Under the constraints of the CAP theorem, CvRDT provide the strongest consistency guarantees for AP settings.

The specifications for the log entry fields are the following:

1. **Prent hash** - SHA-224 hash hexdigest of the target entry.
2. **Timestamp** - a UNIX timestamp declaring the node local time at which the log entry was created. Used solely as informative data, for example by the `ls` command.
3. **Action** - defines the log operation type, enabling common filesystem functionalities.
   - `mkdir`: make a new directory
   - `write`: create or update a file content
   - `delete`: deletes an entry
   - `revert`: reverts a path to some previous state
   - `grant`: enables write permissions to specific key
   - `revoke`: disables write permissions for an specific key
   - `ack`: acknowledge a log branch as valid, needed for maintaining state after key granting or revocation.
   - `link`: a hard link between two entries
   - `slink`: a symbolic link pointing to some path
   - `mode`: give or remove executable file permissions

   Rename operations are implemented with `delete` and `link` actions. Notice that version control is provided

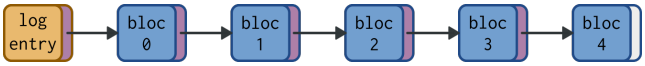

Figure 2: Bloc linking representation

naturally by the monotonic and immutability properties of BaseFS Merkle tree. All history is available, `revert` entries only need to reference a previous entry hash for the path state to be reverted.

4. **Name** - designates the name of the directory, file, link or key. Name size is limited to 256 characters, like most UNIX filesystems. Paths are constructed using this names.

5. **Size** - indicates the size of the content file in bytes. This is a performance optimization that avoids reading the entire file every time an `ls` operation is performed.

6. **Content** - depending on the action this field can contain:

   - `write`: SHA-224 hexdigest of the first block content
   - `grant` or `revoke`: Base64 encoded EC public key
   - `slink`: target path, could be any path, not restricted to BaseFS filesystem
   - `link` and `revert`: target entry hexdigest SHA-224 hash
   - `mode`: file permissions in octal notation

7. **Key fingerprint** - public key fingerprint used to sign the log entry.

8. **Signature** - ECDSA (Elliptic Curve Digital Signature Algorithm) signature of the log entry in base64 encoding.

Write Permissions.

`grant` and `revoke` entries are used for directory-based permission management. A `grant` entry gives a public key permissions to write into a directory and all its sub-directories. Since all log entries are cryptographically signed by its author, BaseFS nodes are able to ignore log entries that do not satisfy required permissions.

BaseFS is a self-certified filesystem, a trust chain can be built only by trusting the filesystem root key, owned by the node that first bootstrapped the filesystem.

Special considerations are needed when revoking keys. The user doing the revocation must acknowledge (`ACK` entry) all related leaf entries, otherwise leaf entries with a now invalid key will be ignored.

Log blocks.

File content is divided into chunks called blocks. As represented in figure 2, blocks form a hashed linked list. A log entry points to the first block, and each block references the next block by its hash. An empty hash is used to signal end of content.

By addressing blocks by their hash the block list is tamper resistance and avoids deduplication. With the convenient side effect of saving disk space and bandwidth on copy operations.

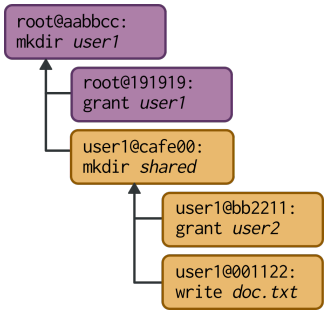

Figure 3: Conflict-free view representation of figure 1 log

3.2 View

Systems that allow replicas to diverge must have a way to eventually reconcile two different states. As a CvRDT, conflicts at the log level are not possible. However, concurrent operations on the same path can create conflicts at the file system level. The log can be seen as a tree of blockchains[4], where every filesystem path represented on the tree has an associated blockchain. Similar to Bitcoin, branches on these blockchains are considered conflicts, and only one branch can be valid at any given time. The *view* is responsible for resolving these conflicts by deciding the valid branches of the log tree.

The adopted strategy for conflict resolution is similar to Bitcoin's *proof-of-work*[17] in the sense that global consensus is not achieved by coordination, but by applying deterministic rules to our Merkle tree. The *view* uses the self-certified properties of the *log* to build a 3-step rulebook for conflict resolution that enables distributed consensus based on *proof-of-authority*.

1. Choose the branch whose contributors have a higher key on the filesystem hierarchy (log entries with incomplete files are ignored until completed). (*proof-of-authority*.

2. If equal, select the branch with more contributors. More nodes agree on the same branch.

3. If equal, select the branch with a higher root hash. Unambiguous, there are no equal hashes.

Users with keys higher on the hierarchy have control over greater portions of the filesystem. Key position in this hierarchy should match the responsibility a user has on the system, its creator being the one with ultimate power.

A consideration when granting higher permissions to an existing key; related conflicting branches may have been resolved by scoring on higher hierarchy, but with an increase on authority this balance may change. Acknowledging the current "wining" branch is required for maintaining state.

3.3 Network

BaseFS uses two different protocols for communicating updates to other nodes and maintain all replicas synchronized:

- Gossip protocol - near-real time communication, maintains group membership

- Synchronization protocol - anti-entropy protocol for repairing replicated data, compares replicas and reconciles differences

Replication is asynchronous, changes are performed locally and then sent to the rest of the network. From the performance perspective this means that the system is fast: the client does not need to spend any additional time waiting for the internals of the system to do their work. The system is also more tolerant to network latency since fluctuations in internal latency do not cause additional waiting.

3.3.1 Gossip Protocol

A gossip protocol is a style of computer-to-computer communication protocol inspired by the form of gossip seen in social networks. Provides weakly consistent knowledge of group membership to all participants as well as probabilistic broadcast of events to all members. BaseFS uses Serf gossip library, which is based on SWIM, Scalable Weakly-consistent Infection-style Process Group Membership Protocol[2]. Unlike traditional heart-beating protocols, SWIM separates the failure detection and membership update dissemination functionalities of the membership protocol. Processes are monitored through an efficient peer-to-peer periodic randomized probing protocol. Both the expected time to first detection of each process failure, and the expected message load per member, do not vary with group size. Information about membership changes, such as process joins, drop-outs and failures, is propagated via piggybacking on ping messages and acknowledgments. This results in a robust and fast infection style of dissemination.

BaseFS uses Serf for a) membership maintenance and b) broadcast of new log objects to the group members. For broadcasting events Serf uses discrete UDP datagrams. UDP is message oriented without ordering, reliable delivery, retransmission nor flow control performed by connection oriented protocols like TCP. It also has the limitation of how much information can be sent by a single event. Specifically, Serf allows event payloads as big as 512 bytes. A conscious effort has been made in order to ensure BaseFS log objects do not exceed this capacity. Figure 4 shows how BaseFS assembles log entries into Serf event payloads, with key optimizations being:

1. The hash function of choice is SHA-224, the smallest SHA (28B) considered secure[1].

2. Use of elliptic curve cryptography with 192 bits key size (equivalent to a 2048b RSA key[2]). Keys of this size produce 48 Bytes signatures.

3. Encoding of the file size value is limited to 6 bytes, restricting the maximum file size to 2 PiB.

4. Binary over text representation of the entry fields. Using fields of fixed length where convenient and offset bytes for more variable fields. A byte can delimit up to 255 bytes, pairing file name length with other modern filesystems.

[1] `https://en.wikipedia.org/wiki/SHA-2#Comparison_of_SHA_functions`
[2] `https://tools.ietf.org/html/draft-ietf-msec-mikey-ecc-03`

To have an approximate idea about the number of gossip messages generated under typical cloud management workloads, figure 5 plots a histogram of the number of messages used for replicating the entire `/etc` directory of a typical Linux box. Including directories, files and symbolic links. Linux `/etc` directory contains the system configuration and can be a good representative of an actual large distributed system configuration. Using any of the tested encoding methods, 0.987 of `/etc` content can be disseminated using at most 10 Serf events per file.

Figure 6 shows the measured time each encoding method takes to process `/etc` files. Bsdiff4, a tool for creating and applying patches to binary files, is perhaps most appropriate method for dynamic configuration. Not only initial patches are comparable in size to other popular compression methods, but the real advantage comes on subsequent file updates. Binary differences between updates are likely to be very small, requiring only one Serf event.

3.3.2 Synchronization Protocol

While gossip produces the initial spread of information, a full state synchronization protocol is run infrequently in order to guarantee delivery with probability 1, update nodes after being partitioned and bootstrap nodes joining the system. Additionally, because the number of blocks sent through the gossip layer can be limited by configuration, a mechanism to spread remaining blocks is needed. This protocol is different from Serf *full state sync protocol* in two ways. It is not limited to the *nth* most recent events and it is optimized with knowledge of the underlying *log* data structure.

In order to make the information exchange during replica synchronization efficient, the sync protocol uses traditional Merkle trees. Data is hashed at multiple levels of granularity and nodes can quickly identify divergent parts of the data. The Merkle tree is built conforming to the filesystem hierarchy. Each path hash is computed recursively, using the XOR of its sub-paths as well as its own related entries. The root path is the XOR of all log entries. The protocol communication is an iterative process, walking and expanding paths with a mismatching hash. Nodes will detect divergence interchanging log entries and blocks until fully synchronized.

The synchronization protocol is a text-based streaming protocol. Using new line character as log entry delimiter, spaces as field delimiter and encodes binary content in base64, avoiding delimiters to appear out of place. Its alphabet is:

- `ID` - Filesystem identification number.
- `LS` - Path list. Includes all path entry hashes, subpaths hashes and the last-block hash of incomplete files.
- `PATH_REQ` - Path request. Indicates a node is missing an entire path and requests all its content to its peer.
- `ENTRY_REQ` - Entry request. Used by a node to request a missing entry to its peer.
- `BLOCK_REQ` - Idem for blocks
- `ENTRIES` - Contains a log entry. Can be a response to an `ENTRY_REQ` or when a node finds out that its peer is missing some entry.
- `BLOCKS` - idem for blocks

1B	28B	4B	1B	1B	0-256B	1B	0-256B	0-6B	16B	48B
event	parent hash	timestamp	action mkdir,create,update, delete,revert, grant,revoke,ack, link,slink,mode	offset	name mkdir,write,link,slink: path name grant: key name	offset	content write: first block hash grant: EC public key slink: target path link,revert: entry hash mode: mode value	file size @2PiB	key fingerprint	signature EC signature

Figure 4: Log entry contained into a Serf custom event payload

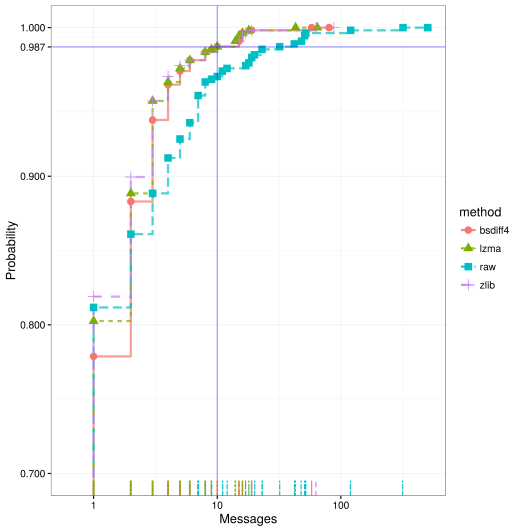

Figure 5: Cumulative histogram of /etc number of messages

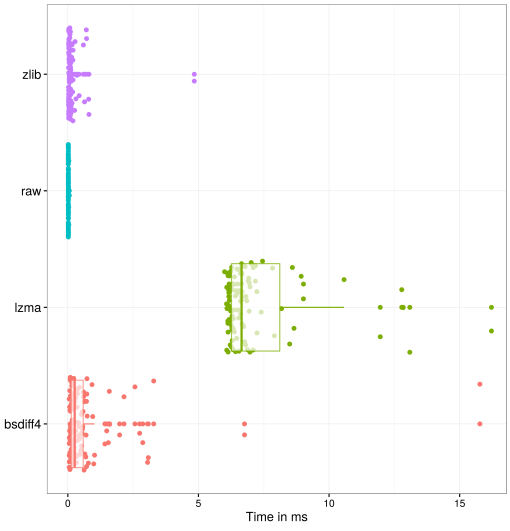

Figure 6: Compression time of /etc files

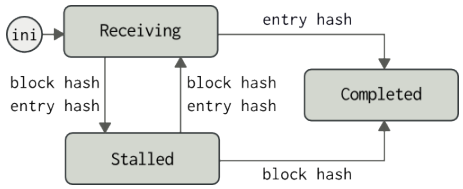

Figure 7: Block state diagram

- `BLOCKS_REC` - A node announces files in receiving state. In case of divergence the peer can apply this hash to the Merkle tree.
- `CLOSE` - Indicates a node is fully synchronize with its peer and communication is terminated.
- `EOF` - Signals end of transmission.

The communication pattern is probabilistic, correlated with the amount of time passed since last contact. Every t seconds, a node chooses a peer i with probability $p_i = t_i / \sum_{j=1}^{1,n} t_j$. Synchronization is initiated by sending `ID` and `LS /` requests containing local state. Things continue from there.

To make dissemination faster for files greater than `MAX_GOSSIPED_BLOCKS`, nodes immediately initiate synchronization with a number of peers specified by a configurable `SEED_NODES`, defaulting to 4.

Block hashes are not included on the Merkle tree. Doing so will make the synchronization protocol very unstable during periods of gossip dissemination. With root hash flapping its value very rapidly. To avoid this effect, as well as preventing nodes to simultaneously retrieve the same blocks from multiple peers, the notion of *block state* is introduce. Files can be in one of the following three states, also represented in figure 7:

- Receiving - indicates a node is being receiving blocks. The sync protocol announces the file as being received, so the other replica can account for it when comparing state.
- Stalled - a file enters this state when no related blocks have been received after some time t. Both, the *entry hash* and the *last received block* are added to the Merkle tree.
- Completed - all file related blocks have been received. The *entry hash* is included to the Merkle tree. In case the previous state was stalled, *last block hash* is removed.

3.4 Filesystem

The filesystem layer provides a well-known API for users and applications to interact with the *view*. The filesystem

interface is implemented using FUSE Python bindings [10]. FUSE stands for Filesystem in Userspace, and allows developers to build virtual filesystems without having to write kernel modules.

The implementation is straightforward, almost limited to *View* operations. Only a couple of optimizations are worth mentioning:

- The *view layer* does not rebuild automatically when new changes arrive from the network. Instead, the *log seek value* is used to check if the *view* is up-to-date with the *log* on each read. A mismatch indicates the log has grown with received entries and a rebuild of the *view* is performed before doing the actual read.
- File `writes` are staged until file `release` is executed. Updates may require multiple `write` operations, BaseFS waits until file `release` to actually write changes to the `view`. Benefits being a) generate a single Bsdiff4 patch and b) summarizes all related `writes` into a single log entry.

3.4.1 Handlers

The naive approach for applications to react to changes is periodic reading (pulling) the state they are interested in. Modern Linux kernels provide support for filesystem notifications via the *inotify* subsystem. Unfortunately FUSE has no support for triggering *inotify* events. BaseFS provides support for executing scripts in response to new log entries in the form of event handlers.

Event handlers are registered at mount time and are invoked in the context of a shell. Can be any executable, including piped executables (such as `awk 'print $2' | grep foo`). Event handlers are executed each time a new log entry is stored. Context for the scripts is given by environment variables such as `BASEFS_EVENT_TYPE` and `BASEFS_EVENT_PATH`.

3.5 Modules Overview

Figure 8 shows the main BaseFS modules and their interactions. BaseFS makes extensive use of concurrency including processes, threads and an event loop. The FUSE interface runs on the main Python thread, as required by its implementation. The Serf agent runs on a separated Python process. Communication with Serf agent is done using Serf's RPC protocol. We spawn an additional thread for the event loop. Implemented with *asyncio*, the event loop handles all the remaining network communication in a non-blocking fashion. Including the synchronization protocol, receiving of gossip events as well as commands sent by BaseFS CLI utility. The event loop thread shares memory with the main FUSE thread, and only a single instance of the *view* has to be maintained, saving substantial memory and computation time.

The modular design allows for easy module replacement. For example, the filesystem module providing a convenient filesystem API to the *view* can be replaced, or complemented, by other interfaces, like HTTP REST or XML-RPC.

3.5.1 CLI Commands

The filesystem API is limited to data operations. For administration and management purposes BaseFS provides a command line tool that talks to the BaseFS daemon via a simple text-based TCP protocol. Some of the commands are:

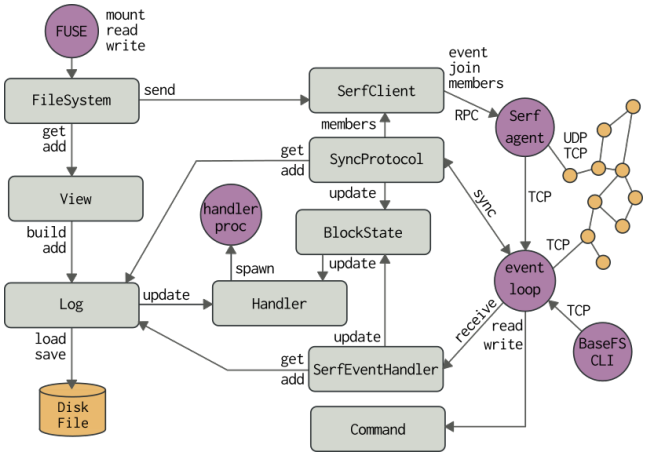

Figure 8: BaseFS modules

- `mount` Mount an existing filesystem
- `run` Run node without mounting
- `bootstrap` Create a new self-contained filesystem
- `genkey` Generate a new EC private key
- `keys` List keys and their directories
- `grant` Grant key write permission
- `revoke` Revoke key write permission
- `list` List all available logs
- `show` Show a log file using a tree representation
- `revert` Revert object to previous state
- `blocks` Show block state of incomplete files
- `members` List group members
- `get` Get log from peer address
- `resources` Monitor resource consumption in real-time

4. PRACTICAL OVERVIEW

This section provides a practical example demonstrating how BaseFS can be used as a community cloud configuration management tool. Our goal would be to provide virtual machines (VMs) as a service on top of a Community Network. VMs as a service is part of a broader category commonly referred as IaaS, or Infrastructure as a Service. Our service will be built and managed by members of the community network whom would take 3 different roles:

- Superuser, an authority figure that the community trust. His key will be the root filesystem key from where the trust chain can be built upon.
- Administrators, they will perform user management tasks like adding or deleting users.
- Users, they will be able to a) contribute hardware machines where the VMs can run and b) allocate VMs on these host machines.

BaseFS organizes data in hierarchical form, similar to that of a traditional filesystem, but with the distinction that write permissions are applied recursively affecting sub-paths. With this consideration, we model our service with the following directory sctructure:

1. `/users/user1/info`
2. `/users/user1/contact`
3. `/users/user1/ssh_keys/key1`
4. `/users/user1/nodes/node1`
5. `/users/user1/apps/app1/vm1`
6. `/handlers/node.sh`

Root superusers are the only ones allowed to write into the root directory (`/`), they have to create the `/users` directory and grant write permissions to all administrators. Administrators are now able to add users by providing the following directory skeleton for each user:

1. `info` file containing user information such as username, this information can not be edited by the user.
2. `contact` file with information that the user is allowed to update.
3. `ssh_keys` directory containing user's public SSH keys, needed for authorizing access to VMs. This folder is writable by the user.
4. `nodes` directory where the user defines its contributed host machines.
5. `apps` directory where the user can create applications with required VMs.

Following is how `node1` and `vm1` configuration files may look like. They are in json format, specifying all required information needed for our service to operate.

```
# node1
{
    "name": "node1",
    "memory": "32GB",
    "disk": "600GB",
    "cpu_units": 10000,
    "state": "active",
    "vm_types": ["debian-8-amd"],
    "zone": ["barcelona", "spain"],
}
# vm1
{
    "name": "frontend-01",
    "app": "app1",
    "node": "node1@user1",
    "type": "debian-8-amd64",
    "state": "started",
    "memory": "1GB",
    "disk": "20GB",
    "cpu_units": 200,
}
```

BaseFS runs on both, user and host machines. Users create the configuration that hosts machines react upon. All changes have an author, and the entire history can be reviewed using `log` command.

BaseFS only requires Python3 and libfuse [22], and it can be installed on any Linux system with a single command. Once installed, we have to generate an EC key pair, this key will be the root key of the new `communityvms` filesystem. Filesystems can be referred by their name. For convenience, default TCP and UDP port numbers are based on that name, so they don't need to be explicitly provided. Next, we bootstrap the filesystem providing its name and the public IP address of the first machine that will run the filesystem, so other nodes can connect and retrieve the log. The filesystem can now be mounted and initialized. Following the command sequence that can be used for performing all aforementioned steps.

```
pip3 install basefs
basefs keygen
basefs bootsrap communityvms -i 10.12.32.12
basefs mount communityvms /mnt
mkdir /mnt/communityvms
basefs grant /tmp/admin1.pubkey /mnt/communityvms
```

Once a filesystem is mounted the node periodically tries to join the group by connecting with bootsraping nodes, 10.12.32.12 in our case. BaseFS provides a `get` command for easily distribute a filesystem to new nodes.

```
basefs get communityvms 10.12.32.12
basefs mount communityvms /mnt/communityvms
```

Nodes hosting VMs have to mount the filesystem specifying a handler that takes care of creating and deleting VMs on that specific node. As a good practice handlers should be stored into `communityvms` filesystem, so upgrades of these scripts are automatically deployed to all nodes.

```
basefs mount communityvms /mnt/communityvms \
    --handler /mnt/communityvms/handlers/node.sh
```

Handlers are idempotent scripts responsible for maintain the node system on the declared state. The following code snippet shows a high level implementation of `node.sh` handler, that creates, updates and deletes VMs when needed.

```
declared_machines = get_declared_machines()
deployed_machines = get_deployed_machines()

# Update and delete VMs
for machine in deployed_machines:
    declared = declared_machines.pop(machine)
    if declared:
        machine.update(declared)
    else:
        machine.destroy()

# Create new VMs
for declared in declared_machines:
    machine = declared.deploy()
    machine.update(declared)
```

We have only covered the basics in this section, other common functionality desirable for community cloud management has been omitted. Features like, advanced allocation policies based on geographical zones (not individual

machines), resource accountability, incentive mechanisms to avoid tragedy of the commons, syntax validation of configuration files, monitoring or auto-scaling. In our example, BaseFS only provides a replication layer for cloud configuration, although it can help implementing more advanced cloud management functionality.

5. EVALUATION

In this section an evaluation of the BaseFS network properties and IO performance is presented. For the validation of log tree conflict resolution and permissions the reader can refere to the unit and functional tests shipped with BaseFS source code[3].

All test scenarios have been fully automated for easily reproducibility. We have developed our own test suite. The test suite has support for virtual environments based on Docker containers, as well as support for deploying and running experiments on Community-Lab testbed[20]. Docker builds on top of the Linux kernel resource isolation features to provide operating-system-level virtualization. Community-Lab is a Community Network Testbed by the CONFINE project that provides a global facility for experimentation with network technologies and services for community networks.

The machine used for running virtual experiments is an Intel(R) Core(TM) i7-4500U CPU @ 1.80GHz, with 4 cores and 7GB of memory.

5.1 Network Evaluation

The network evaluation is separated into two phases that determine a) how constrained network characteristics affect BaseFS convergence time and b) how BaseFS behaves in a real Community Network environment. In this evaluation we want to validate whether BaseFS tolerates worst case conditions expected for a congested wireless link network and characterize its performance over nominal conditions. Convergence time is perhaps the most important characteristic to measure on an eventually consistent datastore, it determines the time it takes for the whole system to be in a consistent state.

Docker containers use virtual Ethernet devices connected to a virtual bridge. All nodes are at one layer 2 hop between each other. TC (Linux Traffic Control) is used for configuring the kernel network scheduler and shape the traffic characteristics of the virtual network. Each experiment is performed on a group of 30 nodes. For each experiment a new BaseFS log is bootstrapped. Nodes get and mount this freshly created BaseFS filesystem. Group members are given a few seconds to find each other. We simulate configuration updates by copying a set of pre-created files into one of the nodes BaseFS mounted partition. Then we measure the time it takes for the configuration file to propagate to the rest of the group. We monitor the number of converged nodes in real time, so the experiment can advance as soon as all nodes have received the updates.

5.1.1 Prelude: Parametrization

Before performing the evaluation we will choose the value of some important BaseFS parameters and environment conditions. In particular we want to establish a limit on the *number of blocks disseminated by the gossip layer*, a good value for the *full state synchronization execution frequency* and which is the *maximum number of Docker containers* we can run without significant CPU contention.

[3] `https://github.com/glic3rinu/basefs/`

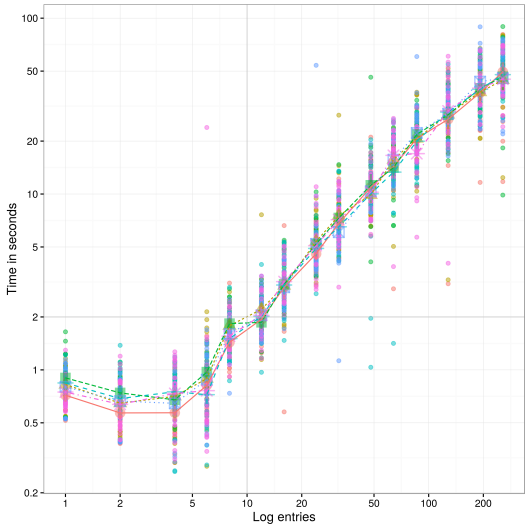

Figure 9: Gossip convergence with variable number of gossip messages

Maximum gossiped blocks.

Gossip capacity is limited by available bandwidth and CPU cycles for generating and processing messages. Under high update load, a gossip protocol may not be able to send all updates required to reconcile differences between peers. Updates would take arbitrary time to propagate as the gossip channel gets backed up. [23]

Sending large files through a gossip channel is inefficient. For establishing a good limit on the number of blocks sent by the gossip layer we have generated a collection of files that produce from 1 to 256 gossip messages. The measured time required for a group of 30 nodes to converge is shown on figure 9.

The gossiped blocks limit for our experiments is set to 10. Being a good compromise between mean convergence time (2 seconds, figure 9) while including a large amount of potential files that can be sent (0.987% of */etc* content, figure 5).

Synchronization interval.

The frequency at which the synchronization protocol is executed determines the convergence time of the group and how much network traffic is required. Measures with different intervals have been done and summarized in figure 10.

We chose **20 seconds** as the default time interval for the synchronization protocol. It is a good compromise between traffic load (2.5Kbps) and convergence time (30 seconds on average), beyond this point traffic load increases exponentially as convergence time remains linear. In any case, only 0.013% of `/etc` content are files big enough (>10 blocks) to fully depend on the synchronization protocol for their replication.

Number of Docker containers.

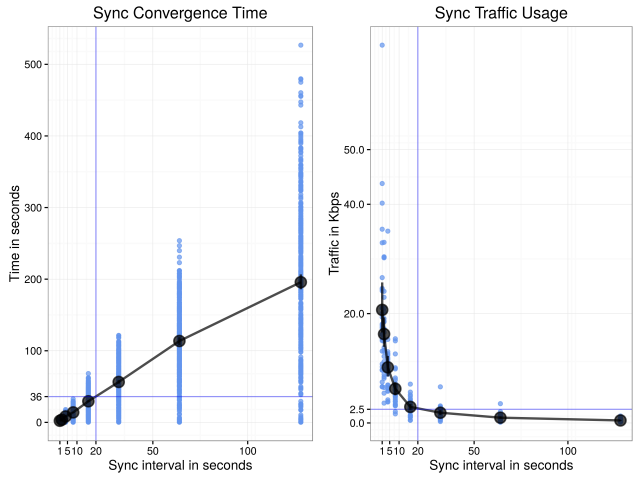

Figure 10: Full Sync protocol convergence with variable execution interval

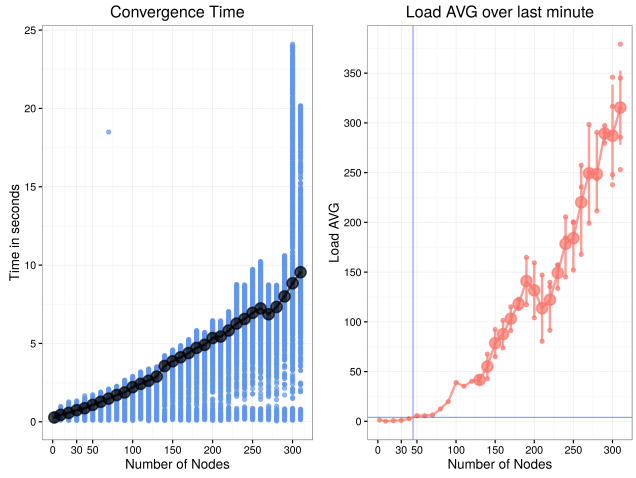

Figure 11: BaseFS Docker scalability

CPU contention is what effectively limits the maximum number of BaseFS nodes that can be emulated on a single machine without compromising measurements. Figure 11 shows the *1 minute load average*[4] of the system while performing 20 writes (separated by 3 seconds) on various group sizes. Writes are crafted to generate predetermined amount of gossip packets, simulating the workload of upcoming experiments. Since our machine has 4 cores, the system is overloded starting from 50 containers and hitting swap at 300. We finally choose a conservative group size of 30 nodes, since the computer is also used by other tasks besides running experiments.

This scalability test has uncovered two caveats of our virtual environment. First, the default value for the neighbor table garbage collector thresholds in the system were set too low, producing overflows on the ARP table[5]. Another problem of tearing up and down hundreds of Docker containers

[4]Number of jobs in the run queue or waiting for disk IO averaged over 1 minute

[5]`https://github.com/hashicorp/serf/issues/263`

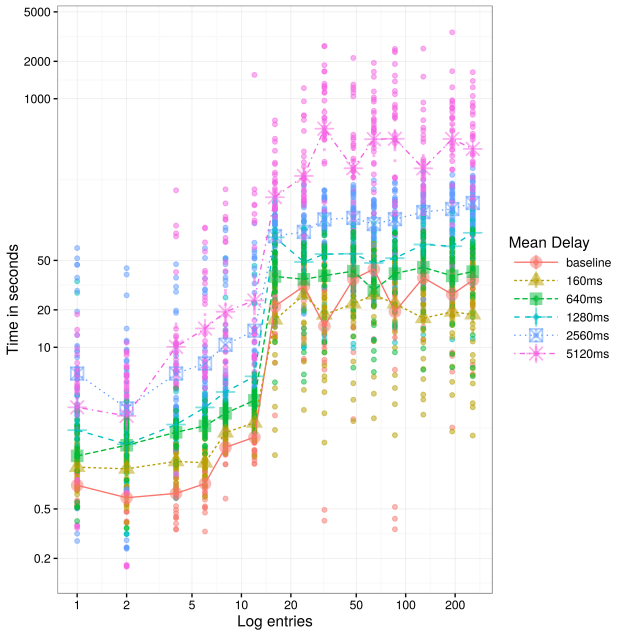

Figure 12: BaseFS under variable delay

is running out of IPv4 addresses because of a Docker bug[6]. Restarting Docker before each experiment solves the issue.

5.1.2 *Delay Effects*

Figure 12 shows the measured convergence time of operations with different number of log objects (1 entry plus n-1 blocks) and delay distributions on a group of 30 nodes. The delay distributions are created using *TC netem discipline* (e.g. `netem delay 100ms 20ms distribution normal`). Standard deviation is kept proportional to 20% of the mean.

Serf is configured to use the WAN profile with a `ProbeTimeout` of 3 seconds, causing nodes to be reported as failed under latencies greater than 3 seconds. Because of the probabilistic properties of the normal distribution Serf is reporting failed nodes starting from 1280 ms mean delays, seriously impacting BaseFS convergence time. However, given enough time, the group is able to converge even with mean delays as large as 5120 ms. Remember that we have selected a *maximum gossiped block* of 10, convergence of writes with more than 11 log objects is totally dependent of the sync protocol, hence the pronounce change on the middle of the plot.

5.1.3 *Packet Loss Effects*

Figure 13 plots the number of converged nodes under different packet loss conditions. Increments of 10% packet loss with 25% of constant correlation are emulated with `TC netem`. For example, `netem loss 30% 25%` causes 30% of packets to be lost, and each successive probability depends by a quarter on the last one. This probability is formally defined as:

$P_n = .25 * P_n - 1 + .75 * Random$

Serf WAN profile sets `GossipNodes` to 4 nodes, causing messages to be gossiped only to 4 random peers. Gossip

[6]`https://github.com/docker/docker/issues/14788`

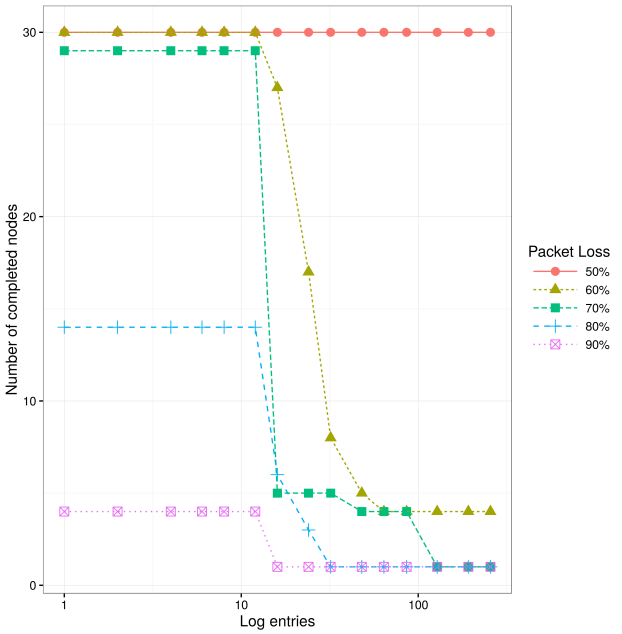

Figure 13: BaseFS completed nodes under variable packet loss

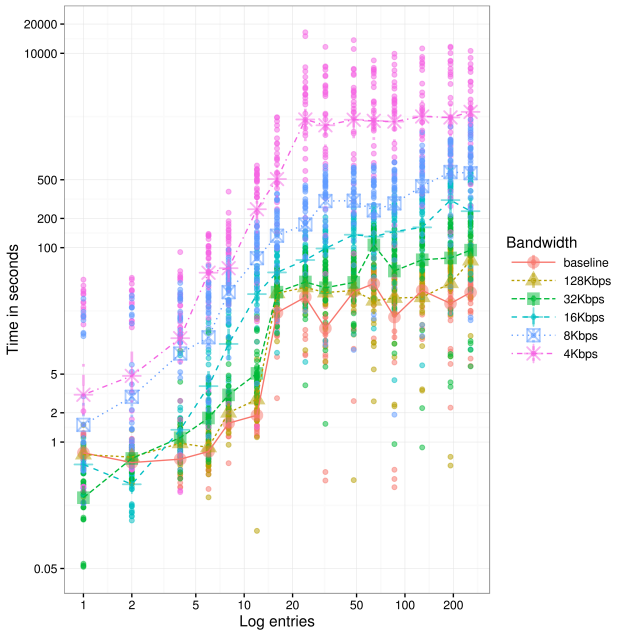

Figure 14: BaseFS under variable bandwidth

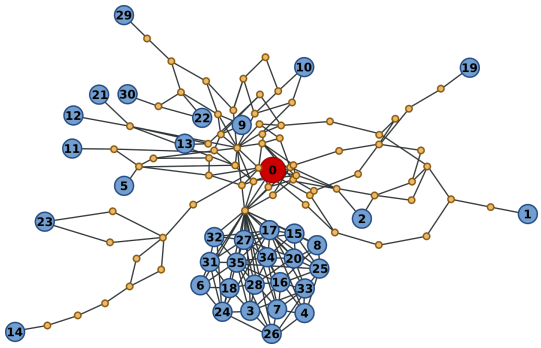

Figure 15: Community-Lab slice network topology

messages are transported over UDP, without retransmission. Lost messages cause the gossip layer to report nodes as failed, also affecting the synchronization protocol, since only alive nodes are contacted. 0.5 chance of packet loss is the tipping point where the system fails to converge. Particularly when block dissemination is not performed by the gossip layer (more than 10 blocks). In this case the number of gossiped messages is not enough to sustained a reliable group member list, and the system rapidly degrades as time passes. By increasing Serf's `GossipNodes` and tunning `SuspicionMult` and `IndirectChecks` we can improve the chances of successful gossip message delivery and the chances of detecting alive nodes, alleviating the substantial effects produced by heavy packet loss conditions.

5.1.4 *Bandwidth Limitations Effects*

Figure 14 shows the measured convergence time of several operations on different bandwidth constrained settings. Hierarchical Token Buckets (HTB) queuing discipline is used to emulate various link data rates, for example `htb rate 32kbit`.

Our measurements are consistent with Serf's analytical bandwidth estimate[7] of 175 kbps/node per message. Bandwidth limitations up to 256 kbps do not have significant impact on the convergence time. Even at low data rates, such as 32 kbps, we only observe a mild 20% time increase respect to baseline. Bandwidth has to be limited to at least 16 kbps to saturate the gossip layer and observe an exponential increase on gossip convergence as the experiment progresses. The sync protocol (+11 log objects) has a more stable progression, as its traffic pattern is less dynamic than gossip.

5.1.5 *BaseFS Under Community-Lab*

We have instantiated a Community-Lab slice consisting on 35 slivers with public IPv4 connectivity between them. To install required BaseFS dependencies we deployed an APT (Debian package manager) and a PIP (Python package manager) proxies over the management network[8]. To ensure all sliver's clocks are properly synchronized we also deployed our NTP server over the management network (NTP traffic is filtered on CONFINE's public network).

Figure 15 shows the public IPv4 network topology of the 35 node slice, uncovering an unfortunate cluster of 20 nodes connected to the same collision domain. Figure 16 shows the hop and latency distributions of the slice.

Figure 17 shows the measured convergence time of a simulated workload on Community-Lab. The workload consists on 560 writes. 60% of which produce one log object, 28% 2 obbjects and 3, 5 and 17 objects are produced 3.5% of the time each. The experiment is replicated using Docker containers for reference.

An evenly distributed traffic consumption throughout the

[7] `https://www.serfdom.io/docs/internals/simulator.html`

[8] `https://wiki.confine-project.eu/arch:management-network`

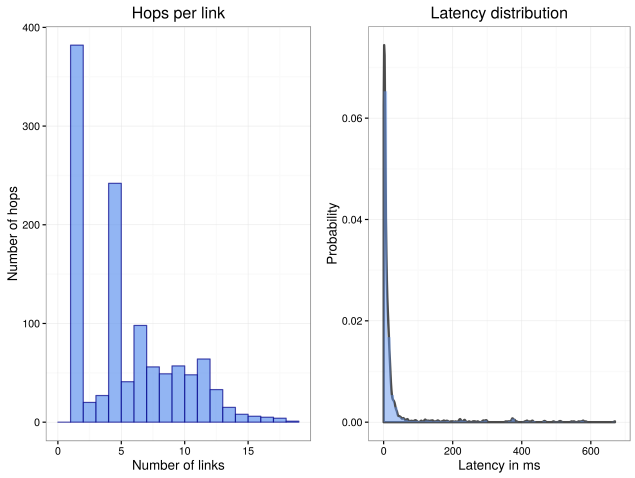

Figure 16: Community-Lab slice characterization

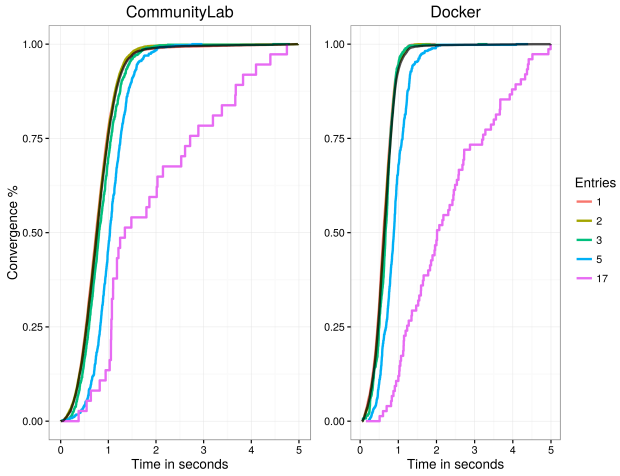

Figure 17: BaseFS convergence time

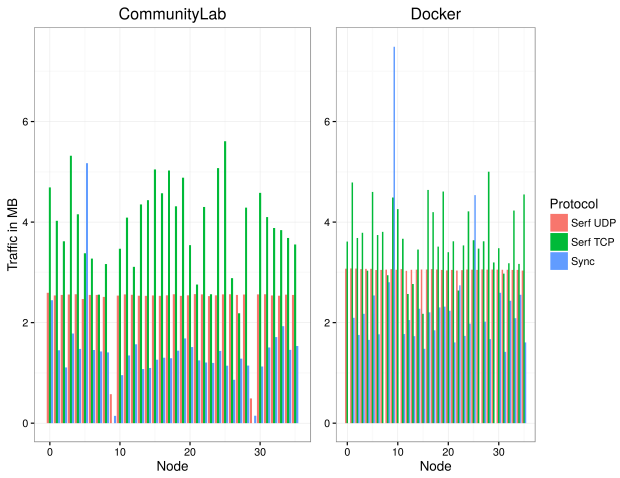

Figure 18: BaseFS outgoing traffic distribution

nodes is a quality expected from any peer-to-peer system. Figure 18 shows the outgoing traffic distribution measured during the experiment. The poor traffic contributions of nodes 9 and 29 are due to an error on Community-Lab testbed. Both nodes did not receive a public IP address, but a private one, lagging behind a NAT. Contact initiated by other nodes is not possible, but NATed nodes can still receive log entries by means of the sync protocol. We didn't intend to have NATed nodes, but this brings the opportunity to validate that BaseFS can deal with an small number of them.

5.2 File Operations Performance

In this section we compare BaseFS to a more traditional filesystem (EXT4). The experiments roughly show how file updates affect read and write performance at the filesystem level, while having a known filesystem like EXT4 to help putting results into perspective. The experiment consists on copying up to 30 times the entire content of the `/etc` root directory (files, directories and symbolic links), a workload designed to hurt Basefs Bsdiff4 usage. Notice that this performance test only involves a single node, the performance of a group is the aggregated IO from all the nodes.

The `/etc` directory of the testing machine contained:

- 2512 files
- 1350 symbolic links
- 462 directories
- 22 MB of data

Bear in mind that we are comparing a kernelspace filesystem (EXT4) with a userspace virtual filesystem that requires executing complex algorithms on top of cPyhton, with the additional FUSE layer and the added cost of having to context switch into kernel mode for performing system calls.

5.2.1 *Read Performance*

Starting from a fresh log file, the entire `/etc` directory is recursively copied into BaseFS mounted directory on each round. Then two reads are performed, the first has to compute the binary difference of every previous version, but the second is cached. We do the same with an EXT4 filesystem stored on a SATA drive. In this case, however, we flush the cache `sync && echo 3 > /proc/sys/vm/drop_caches` and perform the first read as cold as the BaseFS one.

As expected, cold read performance is linearly affected by the increasing number of patches required to apply for obtaining the most recent version of the content of each configuration file. However, a cached BaseFS reads are faster than uncached EXT4 reads, being able to read the entire filesystem clocking at about 2 seconds.

5.2.2 *Write Performance*

Figure 10 shows how BaseFS write performance compares to EXT4. We can see how in each additional recursive copy of the `/etc` directory into the BaseFS partition increases the cost consistently. Apart from writing to the log file, BaseFS calculates the binary difference of each file and computes the conflict-free view of the filesystem. This process can be greatly optimized. However, cloud configuration is about changing small bits of information, without a great concern about the performance of massive write operations.

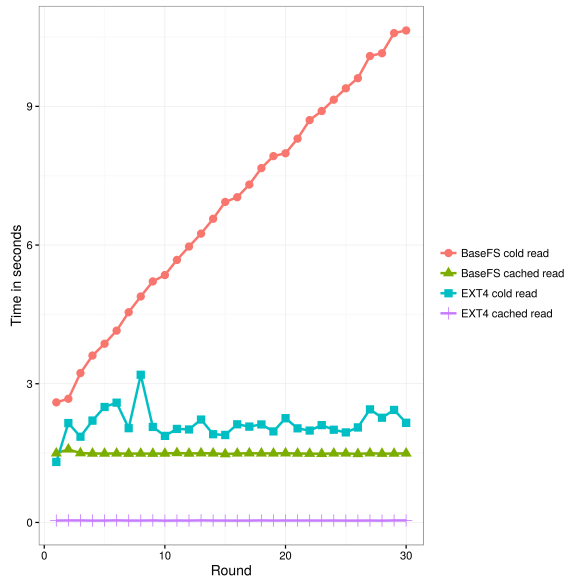

Figure 19: BaseFS vs EXT4 filesystem read performance

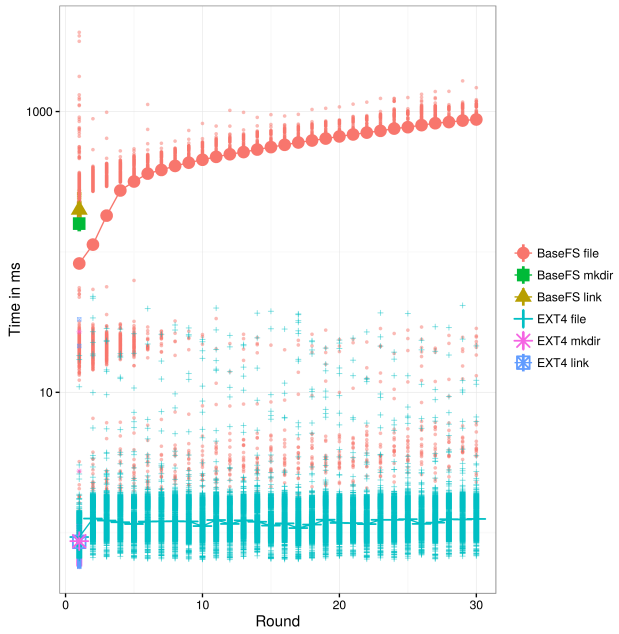

Figure 20: BaseFS vs EXT4 filesystem write performance

Cache invalidation is a hard problem to tackle and is effectively limiting what we are able to cache without paying a great cost on implementation complexity. For one, the conflict-free view of the entire filesystem is recomputed on reads that come after writes. On the other hand, the file content is also invalidated on a write operation and the binary difference has to be computed using all the BSDIFF4 patches that have been generated since file creation, increasing the cost on each update.

We have made the choice of using BSDIFF4 binary deltas on the grounds that write-intensive workloads are not expected for a cloud configuration tool and a faster convergence time (less messages to gossip) is a more desirable characteristic.

6. CONCLUSIONS AND FUTURE WORK

Existing solutions for dynamic configuration management, such as etcd, Zookeeper or Consul, are based on strongly consistent and centralized replication. Making them hard to scale, not available under network partitions and complex to operate. For some situations, eventual consistency is sufficient. With this weaker consistency requirements we devise BaseFS, a new replication system that is peer-to-peer, scalable and simple to deploy and operate. Attributes particularly interesting for community cloud environments.

We have seen how a distributed infrastructure as a service can be built on top of BaseFS. While other solutions would have to secure communications and share secrets, BaseFS can work on the open, making deployment remarkably simpler; install, get and mount. Our measurements also show fast convergence times, typically under a second, and proves it can work under heavily constrained network conditions, with sustained packet loss, large latencies and low bandwidth. Even though we have not shown measurements specific to group size scalability, we believe that the system has the potential to scale to thousands of nodes, as the main bottleneck would be Serf gossip layer for which there are reports of 20K node deployments. File IO performance measurements, although less compelling, are just good enough for cloud management workloads.

Although current design and implementation has proven effective for cloud configuration, the lack of an existing generalized solution with similar characteristics motivates considering what changes are required to make BaseFS a generalized replication service. BaseFS lack of first-class support for files greater than a few hundred MB is a fundamental problem. Important considerations on this regard are:

- **Basdiff4 based encoding**. Bsdiff4 is quite memory-hungry, requiring up to $max(17 * n, 9 * n + m) + O(1)$ bytes of memory, where n is the size of the old file and m the size of the new file. **Multiple encoding methods** should be supported, they can be specified by configuration or perhaps dynamically chosen depending on file characteristics.

- **Hash-linked block list**. Nodes can not know in advance all the block hashes, only the next from the last valid block receive. An approach that provides the block manifest in advance (figure 21) can make block dissemination more efficient and better tolerant to DDoS attacks. In this solution log entries contain the *roothash*, the root node of a Merkle tree, that expands until their leafs can hold the whole file manifest.

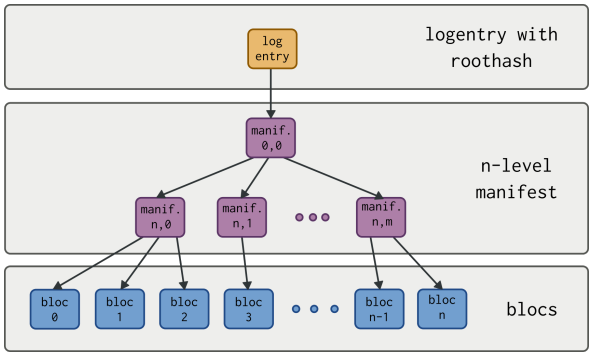

Figure 21: Alternative to block list, based on manifest tree

- **Block dissemination**. Blocks are replicated by means of the synchronization protocol. The sync protocol assumes nodes are always willing to cooperate, but more resources are required as files get bigger, encouraging free-riding. An **incentive mechanism** should be in place in order to discourage non-cooperative behavior. Another issue with the sync protocol is that nodes must receive the complete file before being able to do replication with other nodes. A block exchange protocol like BitSwap, with a **block-market swarm** is a more effective model of replicating large files.

- **Log unbounded growth**. Deleting log entries from the Merkle tree is hard and will required coordinated consensus[14]. Log blocks from deleted or updated files can be garbage collected just by removing them.

BaseFS model is not limited to cloud configuration, it has the potential of being the foundation for new solutions to distributed replication problems where exiting options require nodes to trust each other. Some of the use cases where our model could be attractive are: distributed Dropbox-like applications, system upgrade on distributed systems, shared in-memory database (memcached), mutable P2P file sharing, live documents that self-update when new content is available (encyclopedia or discography), or distributed version control systems.

7. ACKNOWLEDGMENTS

The author would like to thank Leandro Navarro for his time reviewing this work and Ester Lopez for her invaluable help with the `R` code used for producing the plots contained in this document.

8. REFERENCES

[1] zkfuse, zookeeper fuse (file system in userspace). `https://svn.apache.org/repos/asf/zookeeper/trunk/src/contrib/zkfuse/`.

[2] A. M. Abhinandan Das, Indranil Gupta. Swim: Scalable weakly-consistent infection-style process group membership protocol. 2003.

[3] O. Babaoglu and M. Marzolla. Peer-to-peer cloud computing.

[4] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille. Enabling blockchain innovations with pegged sidechains. 2014.

[5] J. Benet. The merkledag, spec for the ipfs merkledag. `github.com/ipfs/specs/tree/master/merkledag`.

[6] J. Benet. Ipfs - content addressed, versioned, p2p file system (draft 3). 2015.

[7] J. Borg. Syncthing, open source continuous file synchronization. `https://syncthing.net/`.

[8] E. Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[9] T. Delaet, W. Joosen, and B. Van Brabant. A survey of system configuration tools. In *LISA*, volume 10, pages 1–8, 2010.

[10] T. Honles. fusepy, simple ctypes bindings for fuse. `https://github.com/terencehonles/fusepy`.

[11] M. Kleppmann. A critique of the cap theorem. *arxiv*, 2015.

[12] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[13] J. Leibiusky. etcd-fs. use etcd as a filesystem. `https://github.com/xetorthio/etcd-fs`.

[14] M. Letia, N. Preguiça, and M. Shapiro. Crdts: Consistency without concurrency control. *arXiv preprint arXiv:0907.0929*, 2009.

[15] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer. Modular composition of coordination services. In *2016 USENIX Annual Technical Conference*, 2016.

[16] A. Marinos and G. Briscoe. Community cloud computing. In *Cloud Computing*, pages 472–484. Springer, 2009.

[17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[18] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*, pages 305–319, 2014.

[19] R. Ranjan, L. Zhao, X. Wu, A. Liu, A. Quiroz, and M. Parashar. Peer-to-peer cloud provisioning: Service discovery and load-balancing. In *Cloud Computing*, pages 195–217. Springer, 2010.

[20] M. Selimi, J. L. Florit, D. Vega, R. Meseguer, E. Lopez, A. M. Khan, A. Neumann, F. Freitag, L. Navarro, R. Baig, et al. Cloud-based extension for community-lab. In *MASCOTS*, pages 502–505. IEEE, 2014.

[21] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt, 2011.

[22] M. Szeredi. libfuse, the reference implementation of the linux fuse (filesystem in userspace) interface. `https://github.com/libfuse/libfuse`.

[23] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 6. ACM, 2008.

[24] B. Wester. Consulfs, consulfs is a fuse distributed filesystem backed by a consul key-value store. `https://github.com/bwester/consulfs`.

[25] O. Yermolaiev. Managing configuration of a distributed system with apache zookeeper, 2014.