Comet: An active distributed key-value store

Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno,
Arvind Krishnamurthy, Henry M. Levy
University of Washington

Abstract

Distributed key-value storage systems are widely used in corporations and across the Internet. Our research seeks to greatly expand the application space for key-value storage systems through *application-specific customization*. We designed and implemented Comet, an extensible, distributed key-value store. Each Comet node stores a collection of *active storage objects* (ASOs) that consist of a key, a value, and a set of *handlers*. Comet handlers run as a result of timers or storage operations, such as `get` or `put`, allowing an ASO to take dynamic, application-specific actions to customize its behavior. Handlers are written in a simple sandboxed extension language, providing properties of safety and isolation.

We implemented a Comet prototype for the Vuze DHT, deployed Comet nodes on Vuze from PlanetLab, and built and evaluated over a dozen Comet applications. Our experience demonstrates that simple, safe, and restricted extensibility can significantly increase the power and range of applications that can run on distributed active storage systems. This approach facilitates the sharing of a single storage system by applications with diverse needs, allowing them to reap the consolidation benefits inherent in today's massive clouds.

1 Introduction

The last decade has seen the rise of distributed storage systems built on loosely coupled collections of autonomous computers. For example, Amazon's S3 [3] provides a key-value storage service for external Web clients. Amazon's Dynamo [17], Apache Cassandra [5], and Project Voldemort [38] provide reliable and scalable key-value stores for company-internal applications (for Amazon, Facebook, and LinkedIn, respectively). On the global Internet, DHTs provided by BitTorrent-based systems, such as Vuze [58] and uTorrent [56], store metadata for millions of clients using peer-to-peer file-sharing applications. And finally, researchers have developed complete file systems on top of untrusted clients in widely distributed P2P environments [2, 14, 44].

Distributed storage systems offer many advantages over their centralized counterparts. For example, a decentralized structure supports scalability; the lack of centralized management enhances automatic load balancing; and the use of replication in a highly distributed environment can improve reliability and data availability. We therefore expect Dynamo-like storage systems to become commonplace as generic application infrastructures in the future, both inside of the enterprise and as shared services on the Internet.

A significant limitation of such systems for generic application support, however, is that different applications have different needs. As an example, each Dynamo application inside of Amazon runs its own Dynamo instance [17], even though a single instance might be logically better and more resource efficient. In our own work on Vanish [25] – a security-oriented DHT application – we needed to make application-specific parameter and policy changes to Vuze (a million-node commercial DHT) in order to harden it against attack. While these changes were conceptually simple, e.g., modifying the storage replication algorithm, deploying our changes took months of work with Vuze's DHT designer. Other Vuze applications may wish to make their own application-specific changes or enhancements, but doing so is neither feasible nor supportable, and it doesn't scale. We believe that with the huge consolidation benefits of shared cloud storage services, either inside or outside of the enterprise, supporting specialization of storage services can have high payoffs in the future.

This paper presents Comet, a next-generation, flexible, distributed storage system, which opens the world of distributed storage to a new set of more complex storage applications. In particular, Comet permits multiple applications to share a single Comet instance, while enabling each application to change the behavior of its storage elements to suit its own requirements. For example, a storage element can make decisions based on its access history, its current number of replicas, the time of day, etc. Therefore Comet can easily support different storage lifetimes, access methods, access control schemes, or replication schemes for different storage-element types, in a way that makes them easy to deploy and test. Using Comet, we can also carry out interesting measurement-based experiments from *within* the DHT.

Comet implements *active storage objects* (ASOs). An active storage object consists of a key, an associated value (an untyped blob), and optionally, a set of simple *handlers*. An ASO's handlers execute as a result of common storage events on the object (such as `get` and `put`) or from timer events that its handlers request. As a result, an ASO can modify its environment, monitor its execution, and make dynamic decisions about its state.

The design of an extensible system for this environment presents a set of interesting design questions. For example, what features should the system provide for ap-

plications and which can (and should) be left out? What is the proper tradeoff between power and safety? How can client nodes be confident that active storage objects will not cause damage or interference? How can we prevent the use of active storage objects to mount a DDoS attack? And overall, how can we extend the storage system without losing its principal characteristics? Our Comet design considers these and other issues.

The remainder of this paper describes our goals, architecture, experience, and evaluation of Comet. To provide concrete insight into Comet's design and potential, we implemented a Comet prototype and used it to create and deploy a set of over a dozen Comet applications. Our prototype leverages Vuze: each Comet instance is an extended Vuze client that can execute Comet active storage objects while also serving as a full participant in the million-node Vuze DHT. Comet applications are written in Lua – a common application-extension language. We modified the Lua runtime to meet our isolation and safety requirements, providing a safe sandbox for handler execution. To test our applications we ran our Comet clients from several hundred PlanetLab nodes and measured their behavior. Overall, our experience demonstrates that a highly restrictive but active distributed storage system can provide significant power to simultaneously support applications with diverse storage needs.

2 Related Work

The concept of extensible systems has been widely explored in the past in several domains. Extensible operating systems have been proposed that support application-specific needs [6, 46, 28]. Active networks allow code to be downloaded along with network data and executed within the network infrastructure (e.g., on routers) to extend network services [60, 54]. Active messages execute a small amount of user code with each message reception [57]. Click explored the design of an extensible router [30]. Database triggers allow applications to define procedural code that is executed in response to database operations [35].

In the context of storage systems, Watchdogs [7] extends the Unix file system, allowing a user-mode process to interpose on file operations for specific files to change access semantics. Several projects have proposed the integration of CPUs and disks to create intelligent disk storage systems that can provide on-board application-specific functions, e.g., for decision support systems, data mining, and image processing [29, 41, 1].

DHTs are increasingly used to support a variety of distributed applications, such as file-sharing, distributed resource tracking, end-system multicast, publish-subscribe systems, distributed search engines, and even data-center applications. Some of these systems (e.g., as CFS [14], i3 [52], and PAST [44]) can be implemented using the traditional put/get interface, but many others (e.g., Mercury [8], CoralCDN [21], Scribe [45], and Bayeux [64]) require customized interfaces and are implemented by altering the underlying DHT mechanisms in significant ways. Our work provides the ability to extend a DHT without requiring a substantial investment of effort to modify its implementation.

Deployed DHTs don't currently offer good semantics and security. However, people do know how to make them consistent [32, 34] and harden then against attacks [11, 16, 48, 26, 59]. The reason DHTs do not currently implement these techniques is that there has not yet been a deployed application that truly needed strong semantics and security. For example, the Vuze design perceived many threats as irrelevant [23] and deployed few defenses against them. However, after the new, more demanding Vanish application was proposed [25], the Vuze DHT responded by embracing a variety of effective security measures. In addition to enabling new applications atop DHTs, we hope to drive the design of these systems towards well-understood, yet unadopted levels of security and consistency.

3 Goals

Comet is a distributed key-value storage system. Like other such systems, a Comet storage object is a <key,value> pair. Unlike previous systems, however, Comet's design facilitates extensible, active storage objects. A Comet application performing a `put` can therefore include, along with a key and value, a small set of *handlers* for that object. The node receiving the `put` stores the handlers along with the key and value, registers the handlers for events that they specify, and executes the handlers when their respective events occur.

Comet's system goals are:

1. *Flexibility.* Comet should be easily customizable to achieve our target functions described below.
2. *Isolation and safety.* A client node running Comet should be protected from the execution of handlers (e.g., an executing handler cannot corrupt the node or use unlimited resources). Handlers should not be able to mount messaging attacks on other nodes.
3. *Performance.* The performance of `gets/puts` on a Comet ASO with null handlers should be the same as on a non-active system, and execution of handlers should have only negligible performance impact.

Isolation and safety are particularly important to our architecture. While Comet can be used in different environments, we designed it to enable wide-scale, outside-the-firewall deployment on autonomous nodes, similar to P2P systems and DHTs. Users downloading Comet must trust it and have guarantees about its behavior. For this reason, Comet enforces four important restrictions:

1. *Limited knowledge:* an ASO is not aware of other objects or resources stored on the same node and has no direct way to learn about them.
2. *Limited access:* an object handler can manipulate only its own value and cannot modify the values of other objects on its storage node.
3. *Limited communication:* an active storage object cannot send arbitrary messages over the network.
4. *Limited resource consumption:* an ASO's resource usage is strictly bounded, e.g., the system limits the amount of computation and memory it can consume.

We are specifically *not* attempting to build a general-purpose distributed programming system, such as PlanetLab [4, 36]; such a system would be unacceptable in our target environment and inappropriate (and unnecessary) for our needs. Rather, our goal is to support relatively simple specializations or actions on simple storage objects. Even very simple specializations can provide a significantly more powerful storage system that enables new types of applications. We therefore take a lightweight and limited approach. As examples, an ASO should be able to perform the following functions:

- *Statistics gathering.* Collect statistics about its use, e.g., by counting the number of `gets` and `puts`.
- *Information tracking.* Log information, such as a list of IPs that performed `get` operations on its value or a recent history of the values it stored.
- *Time awareness.* Take time-based actions, e.g., to make periodic changes to its state or self-destruct after a timer has elapsed.
- *Location awareness.* Make location-based decisions, e.g., choosing where to store based on nodes' network locations.
- *Access control.* Implement simple access control policies on its own.
- *Replication.* Implement different replication policies.
- *Storage system measurement.* Provide insight into the behavior of the distributed storage system as seen by clients executing within the system itself.

As we shall see, the only long-term state available to a handler is its object's value; therefore, any logs, counts, etc., must be stored as part of that value. However, an active object can choose to report only a subset of its stored value record on a `get`, or it can selectively report different values to different callers based on call parameters.

The following sections describe Comet's architecture. In particular, we discuss the tradeoffs required to provide flexibility while also achieving isolation and safety.

4 Architecture and Implementation

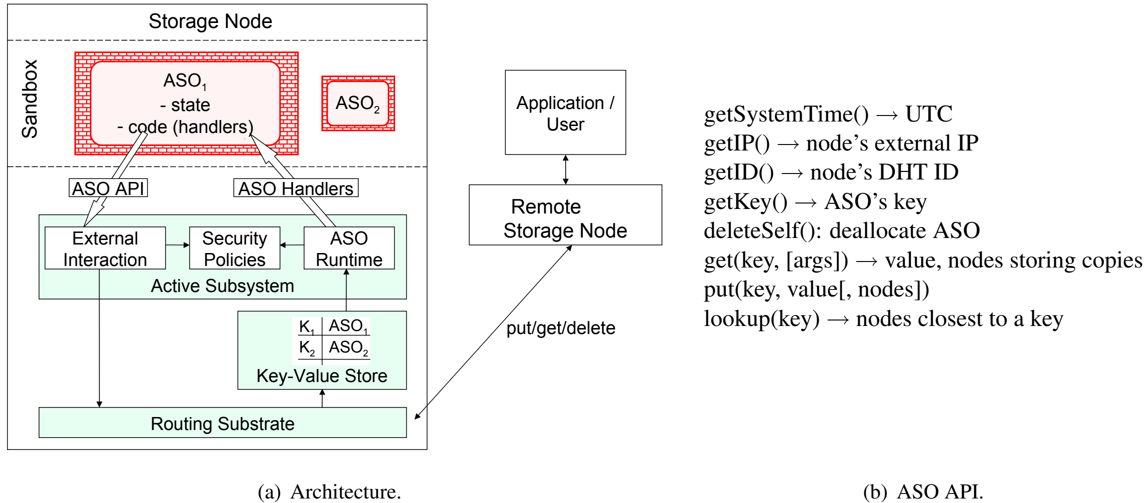
This section describes Comet's active storage architecture and prototype implementation. One could imagine running Comet in various environments, e.g., an inside-the-firewall corporate deployment or a distributed environment with autonomous untrusted nodes. We focus our current architecture and prototype on the latter.

4.1 Architecture

Figure 1(a) shows the high-level architecture of our Comet distributed storage system. The Comet storage system consists of three basic components. First is the *routing substrate* (Figure 1(a) bottom), which implements the value/node mapping, allowing a client to find nodes that store specific data items. In the case of a DHT, for example, the routing substrate typically applies a hash function to the key to compute the IDs of nodes that store the associated value. However, other routing substrates may locate values in other ways.

The second component is the *key-value store*, which maintains a set of key-value pairs on each node. A key-value storage system typically exports a simple `get/put` interface. While existing storage systems store arbitrary, untyped byte strings, the Comet storage system stores *active storage objects* (ASOs). An ASO consists of a key and its associated *state* (i.e., a value, stored as an untyped byte string), along with optional *code* that operates on that state. The code is structured as a set of *handlers* that specify how the object behaves, i.e., how it modifies its state when certain events occur. For example, an ASO's `onGet` handler is invoked whenever a remote client performs a `get` operation to access an object. This handler might perform some simple operation, such as incrementing a counter for the number of gets or appending the client's IP address to a log structure. The counter or the log structure would be stored as part of the ASO's state that can be accessed by the handler.

The third architectural component is the *active runtime system*. The runtime system handles ASO invocations and provides the security policy and execution environment. An application running on a remote client specifies the initial state and handlers for an ASO when initially storing the object via a `put` operation. When a client performs a `get` or a `put`, it can optionally request a cryptographic checksum of the code associated with the target ASO. This can serve as an integrity check that the client's initial `put` is to a key with no associated ASO and that subsequent operations are performed on ASOs created by the application. In most implementations, a Comet node distrusts remote nodes and client applications; therefore, the runtime component of the active subsystem implements and enforces an ASO execution sandbox (Figure 1(a), top). Our Comet prototype uses a language sandbox based on Lua [43] to prevent a handler



(a) Architecture. (b) ASO API.

Figure 1: **Comet Architecture and APIs.** (a) depicts the decomposition of a Comet node into two vertical components - the core Comet code, which is trusted from the node's perspective, and the ASO code which is arbitrary and, therefore, untrusted. (b) details the API exposed to ASOs.

from accessing outside state and to constrain the ASO from consuming too many computational and memory resources on the host. The ASO runtime consults a security policy module, which specifies all execution limits.

While some applications may be satisfied by an entirely sandboxed execution, many would benefit from an ASO's limited ability to interact with or "sense" its environment. For example, to implement the conditional replication scheme we added to Vuze for Vanish, an ASO requires knowledge of the number of replicas in the DHT and the time of day (to enforce the desired minimum replication interval). For this reason, the active subsystem exposes a small API (called the ASO API) to the handlers.

4.2 Active Storage Object API

Table 1 and Figure 1(b) show the handler and ASO runtime APIs, respectively. The handler API supports invocations based on the primary storage functions – `put`, `get` – as well as an `onTimer` handler to be executed periodically (e.g., once every 10 minutes) during the object's lifetime. For example, an ASO could directly implement a custom replication policy in its `onTimer` handler.

The ASO runtime API is the only way for an ASO to interact with its environment outside of the sandbox. Our design supports two types of useful interactions: (1) obtaining information about the local node, and (2) executing various storage system operations. The former category includes functions to obtain the time of day, the hosting machine's external IP address, etc. The latter includes functions to interact with other storage system objects. The ASO API was not designed to be entirely general; rather, our goal was to provide a minimal interface, informed in part by our requirements of security, privacy, and isolation. We tested this interface by implementing and running over a dozen applications on our Comet prototype. Interestingly, we were able to build a relatively diverse set of applications with a surprisingly small interface, which has remained relatively stable through the project. This suggests that a small interface, like the one shown in Figure 1(b), can support a wide variety of applications. Naturally, there are limitations. For example, we explicitly prohibit any direct network-level interactions with remote nodes on the Internet. While this feature might be desirable to certain measurement applications, its DDoS implications would be unacceptable.

onGet(caller[, callbackID, payload]) Invoked when a *get* is performed on the ASO. Returns a value which will be passed back to the caller. Instead of returning a value immediately, the handler could also perform a *put* at the optional *callbackID* sometime in the future. The handler also takes an optional *payload* argument of arbitrary type.
onPut(caller) Invoked upon initial *put* when the object is created. Returns the value that should be stored by the node (e.g., itself or nil).
onUpdate(new_value, caller) Invoked on an ASO when a *put* overwrites an existing value. Returns the value that should be stored, e.g., *new_value* if it should be replaced, or itself if not.
onTimer() Invoked periodically. This handler has no return value. It is used to perform periodic maintenance such as replication.

Table 1: **ASO Handler Calls.**

4.3 Language Based Sandbox

Our Comet prototype focuses on a DHT environment composed of a large number of untrusted autonomous nodes that cooperate to support the distributed active storage system. In this environment, the key challenges include providing a strong sandbox and limiting ASO resource consumption. We briefly describe how our system addresses these challenges using a language based sandbox.

The Comet prototype required an ASO programming environment that reflected our needs for simple extensibility, flexibility, performance, isolation, and safety. To meet these needs, we chose Lua [43], a lightweight and easily constrained scripting language. A dynamically typed, imperative and functional programming language, Lua is most commonly used for coding application extensions. In this context, it lets users add or modify features in video game engines, Web servers, version control systems and other applications (specific examples include World of Warcraft, SimCity 4, Adobe Photoshop Lightroom, and Squeezebox Jive Platform). Several properties make Lua well suited for implementing ASOs. First, it employs a small set of programming constructs (including first-order functions) and a small number of data types (including tables, which are heterogeneous associative arrays). Second, Lua compiles to simple bytecode, which makes it relatively easy to sandbox. Finally, ASOs written in Lua are concise and small when serialized; the Lua ASOs we implemented are all under 1.5KB, about five to ten times smaller than Java equivalents.

Comet represents ASOs as Lua tables that encapsulate both persistent state and the handlers to be invoked on that state. Lua tables can implement basic arrays, associative arrays, or both. While an associative array can contain any name-value mappings, we treat certain associations as handlers. In particular, if the ASO table contains an associative array with the names "onGet," "onPut," "onUpdate," or "onTimer" – and those names are associated with values that are Lua functions – then the runtime invokes those functions when the corresponding events occur. Our runtime system serializes Lua tables into a byte stream for transmission to a storage node on a `put` request.

We made several modifications to the standard Lua interpreter for the Comet runtime system. We sandbox ASOs by removing all but the core libraries from the runtime, leaving only a math package, string manipulation, and table manipulation. As a result, handlers are extremely restricted: they have no direct network access, no system execution capabilities, no thread creation capabilities, and no file system access. We also strictly bound the amount of resources that a handler can consume. For example, the runtime limits both the number of bytecode instructions that a handler can execute and the amount of memory it can consume. If a handler exceeds either of these limits, the runtime terminates its execution.

The Comet runtime exposes a DHT wrapper object to handlers, which allows an ASO to communicate with its environment. The ASO can learn information about the hosting node, including the external IP address and the current system time. It can also perform a restricted set of DHT operations. For example, it can perform `get` and `put` operations on *replicated* copies of its value stored at other nodes. In the API presented in Section 4.2, these operations return values or neighboring node IDs. However, since these operation are slow in the DHT setting and may block for seconds or even minutes, we chose to implement them using function callbacks. Each such operation takes an optional parameter, a function which accepts the result as its parameter. For example, instead of returning a value, a `get` operation takes a function which is eventually passed the result of the operation. The operation returns immediately with no value, and the `get` is actually performed after the ASO execution has completed. While this presents a slightly different paradigm to the user, we think this provides a greater ability to optimize the performance of Comet-based applications.

4.4 Comet Prototype Implementation

We built the Comet prototype on the Vuze DHT, which supports the widely used Vuze BitTorrent client. The DHT is used mainly for distributed tracking of torrents; however it has been used in research as well [27, 25].

Vuze implements the Kademlia routing protocol, in which each node is assigned a 160-bit ID based on the SHA1 hash of its IP address and port. Basic DHT operations (`get`, `put`, and `remove`) take a 160-bit key, perform a lookup to find nodes whose ID is *close* to that key, and then send a read or store RPC to those nodes.

We minimally extended the Vuze interface to conform to Comet's abstract operations. For example, we augmented `get` to allow a caller to pass an arbitrary byte-string argument. This supports a parameterized `get` operation, where the ASO can return different values depending on the parameter (analogous to the semantics for GET in HTTP).

Allowing extensibility in a DHT environment creates challenges, e.g., it has the potential to provide a platform for DDoS attacks. Therefore, in addition to the Lua resource restrictions described previously, we limit DHT communications that ASOs can perform in two ways.

First, we do not allow an ASO to perform operations on arbitrary DHT keys or nodes, but rather only on specific key-node pairs. An ASO may communicate with any of its neighboring nodes that are responsible for replicas of the ASO. We also allow the ASO to communicate with key-node pairs that have interacted with it in the past, once for each such interaction. To enable this functionality, we extended Comet requests to include the ID of the requesting node and the ID of a local key contained within the node. If an ASO receives a `get` request with a key ID specified, it gains the capability for a one-time operation on that key to the node that issued the request. The ASO can then either return a value immediately and exhaust its one-time capability, or save that capability for future use. This mechanism allows applications to respond to DHT requests at a future point in time, es-

pecially if the requested data is not currently available. We do not allow ASOs to pass these capabilities between each other as doing so would enable a malicious node to mount DDoS attacks. In Section 5 we discuss signed ASOs, which do not have these restrictions.

Second, Comet imposes rate limits on the number of messages generated by an ASO, either to neighboring nodes storing replicas or to arbitrary key-node pairs that have interacted with it in the past. This prevents misbehaving ASOs from exhausting the bandwidth resources of the Comet nodes hosting them. We discuss these security issues further in Section 7.

5 Applications

This section seeks to demonstrate both the range of storage behaviors that Comet can support and the ease with which those behaviors can be implemented. To do this, we describe several of the active storage applications we have implemented, deployed, and measured on our Comet PlanetLab prototype. We provide code snippets to show how simply these actions can be programmed in our Lua-based ASO environment. In Section 6, we present measurements from some of these examples.

5.1 Customizable Replication

Most DHTs specify a fixed replication policy for stored values, requiring applications to conform to that policy. In contrast, Comet ASOs can provide their own application-specific replication mechanisms, e.g., controlling the replication factor, the replication interval, and the choice of nodes on which the object will be replicated. This flexibility is useful for applications that place varying degrees of emphasis on performance, availability, locality, and security. For instance, a security sensitive application (such as Vanish) might use a small number of replicas and long replication intervals, limiting the dispersion of its objects stored in the DHT. On the other hand, an application that values availability might replicate frequently to a large number of nodes.

Listing 1 shows how an ASO can define a customized replication policy. In this example, the `onTimer` handler wakes up periodically, invokes `lookup` to determine a list of nodes closest to the ASO's key, executes `selectGoodNodes`[1] to identify a subset of nodes that will serve as replicas, and then stores a copy of itself on the selected nodes using `put`. We have also implemented a timer handler that replicates only when the number of existing replicas falls below a certain threshold; this lowers communication overhead and mitigates data harvesting attacks for security sensitive applications, reflecting the changes we made to Vuze after we published Vanish [25].

[1]The Lua code for `selectGoodNodes` is omitted for brevity. It implements an application-specific policy for choosing replicas.

```
function aso:handleLookup(nodes)
    nodes = self.selectGoodNodes(nodes)
    dht.put(dht.getKey(), self, nodes)
end
function aso:onTimer()
    dht.lookup(dht.getKey(), self.handleLookup)
end
```

Listing 1: Smart Replication

5.2 Controlling Data Access

Comet objects can implement various policies that control how data stored in the objects is accessed. We illustrate a few such examples.

Timeouts and Limited-read values: ASOs can be used to implement objects that will be accessible for only a limited, application-specified time. Such objects are meaningful for security applications such as Vanish [25], which provide support for self-destructing digital data by storing cryptographic keys in a DHT.

Listing 2 shows the handler code required to implement application-specific timeouts. Each replica stores a timestamp when the object is created (stored) and then deletes the object after 60 minutes using a timer handler. In addition, the `onGet` handler prevents the object's contents from being accessed after the timeout but before it is deleted by a timer handler.

```
function aso:onPut(value)
    self.timeout = dht.getSystemTime() + 60*MINUTES
    return self
end
function aso:onTimer()
    if (dht.getSystemTime() > self.timeout) then
        -- delete local ASO
        dht.deleteSelf()
    end
end
function aso:onGet()
    if (dht.getSystemTime() > self.timeout) then
        -- delete local ASO
        dht.deleteSelf()
        return nil
    end
    return self
end
```

Listing 2: Timeouts

An ASO can also choose to delete itself after it has been read – providing a "limited-read value" – where each replica can be read at most once. In addition to its use for self-destructing data, limited-read values could be used in settings where objects represent tasks and are deleted once they have been claimed by worker nodes. The object then serves as a synchronizing construct between the task's producer and consumer.

Listing 3 implements limited-read values. When a `get` is performed, the node records the fact that the value has been read. It then propagates the request to every other replica by overwriting them with `nil`. Note that the object does not delete itself immediately, but rather stays around for a while and periodically attempts to delete other replicas to ensure that copies on nodes with transient connectivity issues [22] are eventually deleted. Note also that concurrent `gets` issued to different replica nodes might successfully read the value. In general, as with other distributed storage systems, consistent update of replicated values would require the use of heavy-weight consensus operations. Comet does not currently provide such primitives. ASO handlers do however provide the ability for replicas to detect and correct inconsistencies, e.g., ASOs can compare and reconcile replica contents through periodic invocations of the `onTimer` handler.

```
function aso:onGet()
    if (self.read) then return nil end
    self.read = dht.getSystemTime() + 30*MINUTES
    dht.put(dht.getKey(), nil) --deletes replicas
    return self
end
function aso:onTimer()
    if (self.read) then
        dht.put(dht.getKey(), nil) --deletes replicas
        if (dht.getSystemTime() > self.read) then
            dht.deleteSelf()
        end
    end
end
```

Listing 3: Limited-Read Values

Data Subscription: An ASO can allow clients to “subscribe” so that they will be notified when the ASO receives a new value. In Listing 4, when the subscriber performs a `get`, the ASO saves the subscriber's network location (`callerNode`) and a key that will serve as the subscriber's recipient of the value (`callbackKey`). When a value `update` occurs, the ASO distributes the value to all registered subscribers – the runtime ensures that the ASO distributes these values *only* to clients who have actually performed a `get` on the ASO. In the example shown, the ASO clears its subscriber list after its `put` operations; subscribers must then re-subscribe if they're still interested. Later we will describe an implementation of a scalable publish-subscribe scheme based on this design.

Sensitive values: ASOs can implement various forms of access control policies. For instance, Listing 5 provides read access to the object's value only if the client can present a predetermined password akin to a feature already provided by some DHTs, like OpenDHT [40]. A client provides the password as an argument to the `get`

```
aso.pending = {}
function aso:onGet(callerNode, callbackKey)
    if(self.value) then
        return self.value
    end
    self.pending[callerNode] = callbackKey
    return nil
end
function aso:onUpdate(callerNode, value)
    self.value = value
    for callerNode,key in pairs(self.pending) do
        dht.put(key, value, {callerNode})
    end
    self.pending = {}
end
```

Listing 4: Pub-sub

request.

There are a few issues with the code provided above, especially if it were to be extended to support password-protected updates. A malicious node could claim to store the object but simply serve as a proxy for clients' requests and thereby implement man-in-the-middle attacks. This could be solved by exposing basic encryption primitives to the ASO, like a secure hash function and/or public key cryptographic primitives. For example, instead of passing the plaintext password to the ASO, the client hashes the concatenation of the password with its IP/port, thus the ASO can verify that the request is not being forwarded by a malicious node. The ASO's security can be further strengthened by public/private key pairs, with the ASO storing the public key and clients authenticating themselves by presenting a message signed with the corresponding private key. With these enhancements, a malicious node storing a copy of the object cannot overwrite the contents of other replicas since it doesn't possess the private key.

```
function aso:onGet(caller, callerId, password)
    if (password == ''mypass1234'') then
        return ''Well kept secret''
    end
    return nil
end
```

Listing 5: Password

An application could use multiple mechanisms for controlling data access, e.g., it could use timeouts in conjunction with password-protected access. While Comet does not allow ASOs to register multiple handlers for a given storage operation, the developer can combine all of the desired mechanisms into a single handler. Though this might increase programming complexity, it allows the application developer to control how different mechanisms interact with each other and provides the basis for a predictable and deterministic execution model.

5.3 Measurements and Monitoring

DHT Measurements: ASOs provide a platform for instrumenting and measuring the DHT using the DHT nodes themselves. This enables a more detailed and comprehensive view of the DHT and helps provide accurate estimates of DHT properties such as churn, node lifetime distribution, transient inconsistencies, etc.

For instance, Listing 6 tracks the k closest nodes to the ASO and stores the information it learns as part of the object state. A measurement application can create objects of this type, store them at multiple locations within the DHT, and obtain snapshots of DHT membership by retrieving the objects' contents using `get` operations.

```
aso.neighbors = {}
function aso:handleLookup(nodes)
    self.neighbors[dht.getSystemTime()] = nodes
end
function aso:onTimer()
    dht.lookup(dht.getKey(), self.handleLookup)
end
```

Listing 6: Lifetime

While this measurement could be performed by nodes that are not part of the DHT (as in earlier work [20, 50]), measurements from within the DHT can provide more accurate data. For example, the lifetime measurement could be carried out by a client that interactively crawls the routing tables of the DHT nodes and then uses heartbeat messages to monitor the uptimes of the nodes it learns about. This approach could provide faulty data, however, if the DHT contains firewalled nodes that do not receive or respond to such heartbeat messages.[2] On the other hand, firewalled nodes still communicate with neighbors, for example to replicate values. Therefore, measurements performed from ASOs *within* the DHT can be more accurate, as we will demonstrate later.

Monitoring uses: An ASO can also maintain audit trails, e.g., indicating where it has been stored thus far, who has read or updated the object, etc. Such tasks are particularly useful for debugging and aid in rapid prototyping. For example, this may help a developer to learn whether a new ASO replication mechanism is operating properly. Alternately, logs can also be used for forensics. Listing 7 illustrates a monitoring application that tracks the nodes storing and accessing a value.

This specific implementation comes with a few caveats. Each replica may have a different view of the list of nodes that have stored or read the value. To address this, the experimenter needs to get the union of the lists stored in all the replicas, consolidating them as a post-processing step.

[2] In fact, about half the nodes in P2P DHTs are firewalled [23].

```
replicaIps, hostIps, accessorIps = {}
function aso:onGet(callerIp)
    table.insert(self.accessorIps, callerIp)
    return self
end
function aso:onPut(caller)
    table.insert(self.accessorIps, caller.getIP())
    table.insert(self.hostIps, dht.localNode.getIP())
    return self
end
function aso:handlePut(nodes)
    for i,v in ipairs(nodes) do
        table.insert(self.replicaIps, v.getIp())
    end
end
function aso:onTimer()
    dht.put(dht.getKey(), self, 20, self.handlePut)
end
```

Listing 7: Monitoring

5.4 Smart Rendezvous

DHTs are used for rendezvous in many distributed systems. In P2P file-sharing systems such as BitTorrent, the DHT is used as a distributed tracker either with or as a replacement for a centralized tracker. That is, peers that want to download a particular file use the DHT to identify other peers who are downloading or sharing the file. The downside with current DHT-based distributed trackers, however, is that they result in random overlay connections, as there is no mechanism to enforce more intelligent peer-matching techniques.

With Comet we can address this limitation by using ASOs to track participating nodes, as well as construct peer lists that are optimized for a requesting node. Peers could be matched in order to lower inter-node latencies [33], maximize reciprocation probability based on peer bandwidths [37], or lower ISP costs [62, 12]. We have implemented one such matching scheme that uses the nodes' network coordinates to predict inter-node latencies and provides a list of nearby peers to each joining node. We describe this in depth in Section 6.3.2.

5.5 Signed ASOs

The examples discussed so far adhere to the strict security policy we set out: ASOs cannot perform operations on arbitrary DHT keys or nodes. We now consider uses where we relax this assumption, but require that the ASO code be *signed* by the DHT administrator after manual verification of its security properties. As we will see below, this allows the DHT to deploy new functionality and services by using signed ASOs that access arbitrary DHT locations, but are safe (i.e., they do not enable DoS attacks of targeted DHT nodes).[3] We have considered signed

[3] In some cases, the safety of the ASO code could presumably be verified automatically, e.g., by using sophisticated compile-time analysis;

ASOs in particular as a mechanism that a DHT's developer or administrator could use for testing and evaluation of new features, before they are added to the main-line DHT code.

Recursive Get: Vuze and many other DHTs support iterative routing for key lookups. In this approach, the node performing the lookup is involved in every step of the routing operation, i.e., it identifies the target node by repeatedly querying DHT nodes to find other nodes that are closer to the target key. An alternative is to perform recursive routing, where intermediate nodes on the route pass the lookup directly to nodes that are closer to the key. Iterative lookup provides greater control to the node performing the lookup (e.g., it can control lookup parallelism), but it comes at the cost of increased latency. If both forms of lookup are available, an application would use recursive lookups by default, but fall back on iterative lookups after persistent failures [15].

With signed ASOs it is possible to implement recursive lookups even though the underlying DHT supports iterative lookup by default (as is the case with Chord, Kademlia, and Vuze). The node initiating the lookup creates a `query ASO`, which contains a reference to itself, and a local callback ID where it would like to receive the answer. When the signed ASO is created its `onPut` handler is invoked; the handler queries the local routing table to find a live node that is closest to the target key, stores a copy of the signed ASO on this node, and deletes itself from the current node. This process is repeated until one of the nodes storing the target is reached, and the `onUpdate` handler of the target ASO sends the object's value back to the original node, which initiated the request.

Caching and Hierarchical Publish-Subscribe: This idea can be extended to accomplish both caching and hierarchical publish-subscribe data delivery. For caching, the `onUpdate` handler can be modified to communicate the object not only to the requesting node but also to the intermediate node that conveyed the request. The number of intermediate nodes to which the object is replicated can be determined by gathering and analyzing statistics on object popularity (also accomplished using simple handler code), so that only popular objects are replicated at multiple nodes (as in Beehive [39]). To implement hierarchical publish-subscribe, intermediate nodes propagate a subscription event to the next node in the lookup process only if they haven't done so before and maintain state for subsequent queries routed to them. When a value is published, it is propagated through a dissemination tree so that the communication load is distributed across all intermediate nodes (as in Scribe and Bayeux [45, 64]).

studying this is part of future work.

5.6 Summary

This section described a set of example storage objects that we have implemented using Comet. Through these examples, it should be clear that with very small extensions (on the order of a few lines or a few tens of lines of code), a Comet application can create a wide range of powerful storage object behaviors that would be impossible in existing distributed storage systems or DHTs.

6 Evaluation

We deployed Comet on approximately 200 PlanetLab hosts and evaluated our design in three steps. First, we characterize the resource utilization of the various applications that we developed. Second, we measured microbenchmarks to understand the overheads associated with active storage objects. Lastly we report on our experiences with prototyping applications using Comet.

6.1 Application Characteristics

Table 2 shows resource consumption requirements for our Comet applications. The *Max Instructions* column gives the number of dynamic Lua instructions required to execute the most expensive handler, while *Execution Time* gives the execution time for that handler. Where this value is data sensitive, we provide an estimate based on the expected maximum value. *Code Size* shows the size of each ASO with the minimum amount of data and *Max Size* is the maximum size to which the ASOs might grow for that application. From the table we see that most ASOs execute fewer than 100 Lua instructions and are smaller than 1KB in size.

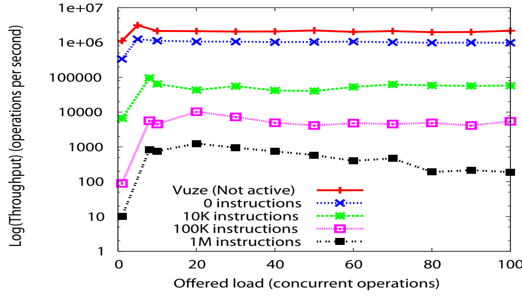
Application	Max Instructions	Execution Time	Code Size	Max Size
Replication	< 10	$4\mu s$	$0.223K$	$< 1K$
Smart Replication	< 100	$6\mu s$	$0.444K$	$< 1K$
Timeouts	≈ 10	$4\mu s$	$0.434K$	$< 1K$
Limited-Read Value	≈ 10	$4\mu s$	$0.553K$	$< 1K$
Sensitive Value	< 10	$4\mu s$	$0.230K$	$< 1K$
Pub Sub	$10,000$s	$54\mu s$	$0.498K$	$100K$
Hierarchical Pub Sub	100s	$6\mu s$	$0.673K$	$1K$
Lifetime (External)	100s	$6\mu s$	$1K$	$6K$/hr
Lifetime (Internal)	< 100	$6\mu s$	$1.776K$	$\approx 3K$
Monitoring	≈ 10	$4\mu s$	$0.971K$	$3K$/hr
Smart Rendezvous	$1,000$s	$14\mu s$	$1.107K$	$10K$
Recursive Get	≈ 50	$6\mu s$	$0.714K$	$\approx 1K$

Table 2: Expected Application Resource Consumption

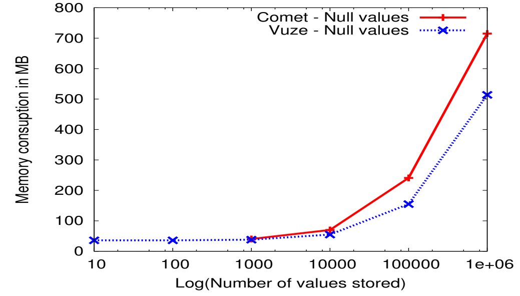
6.2 Performance and Overheads

We report on simple microbenchmark measurements to compare the CPU and memory costs of Vuze and Comet. These experiments were run on an quad-core machine with Xeon processors clocked at 2.67GHz.

Single-Node Throughput. In this experiment, concurrent `get` operations are performed on many values stored in the target node. We measure the throughput of `get`



(a) Single-Node Throughput.



(b) Memory Footprint.

Figure 2: **Microbenchmarks.**

requests that return successfully using a closed feedback loop. All operations are issued locally on the node, so that network latency does not affect throughput.

Figure 2(a) compares the throughput of objects with different ASO execution costs, expressed as the number of Lua bytecode instructions executed per handler. Both Comet and Vuze experience peak throughput when the number of concurrent operations is equal to the number of cores (eight). ASOs with zero instructions per handler are functionally equivalent to Vuze values as they simply return themselves. The peak throughput of Comet ASOs is about 60% smaller than the peak throughput of Vuze (1.4M operations per second as opposed to 3.5M operations per second). This shows the cost of the Comet/Lua execution environment. Previous measurements [49] show that the typical DHT load on Vuze clients in the wild is at most a few hundred operations per second, which makes the additional Comet overhead relatively insignificant in this context. As we increase the computational complexity of the average ASO (1K to 1M instructions per handler), the throughput decreases, but still remains well above the maximum current Vuze workload.

Operation Latency. At the 90th percentile, with maximum throughput (8 concurrent operations in our experiments), a request involving 100 Lua instructions has a latency of about 300 microseconds. For handlers with 1M instructions (two orders of magnitude more than our most compute-intensive handlers), it is 13 milliseconds. The latency for a Vuze DHT lookup is on the order of seconds, therefore the latency imposed by even extremely computationally intensive ASOs is not significant.

Memory Footprint. In this experiment, we store increasing numbers of values in the nodes. For the Vuze nodes, the string "hello world" is stored at different keys, while for Comet nodes we store an equivalent Lua ASO which returns the same string upon a `get` request. Figure 2(b) compares the memory footprint of the Vuze and Comet nodes as we increase the number of stored objects. Again using the median number of values stored per Vuze node (around 400), the difference in memory consumption at this level is negligible (about 36MB for both Comet and Vuze). Long lived DHT nodes can store 10,000s of values, and the highest observed is around 30,000 values [49]. In these rare cases, our overhead relative to Vuze is about 27%, but even then the total memory footprint is still reasonable.

We next consider a workload where Comet object sizes are exponentially distributed with an average size of 10KB. In this case, a node with 500MB can store on average 50,000 values. If we assume an order of magnitude more values per node than in Vuze (4,000 instead of 400), and an order of magnitude larger values (10KB instead of 1KB limit imposed by Vuze), the median node would consume about 80MB (40MB of startup memory costs and another 40MB for the ASOs) in memory.[4]

6.3 Application Experience

We now report on our experiences in prototyping and deploying some of the applications described in Section 5.

6.3.1 Measuring Node Lifetimes

We revisit the experiment performed by Falkner et al. [20] to measure the lifetimes of nodes in the Vuze DHT. This experiment was done by performing random get operations from several Vuze clients in order to gather approximately 300K IPs participating in the DHT. The collection of nodes was then pinged every 2.5 minutes to check for liveness. The authors observed that nearly half the nodes were immediately unavailable after first being detected. One weakness of the methodology employed is that the clients could not differentiate nodes that are unreachable because of NATs from those that have left the DHT. Using measurement nodes that have active communication channels with NATed DHT nodes would help minimize

[4] Vuze and Comet consume about 40MB without storing any values.

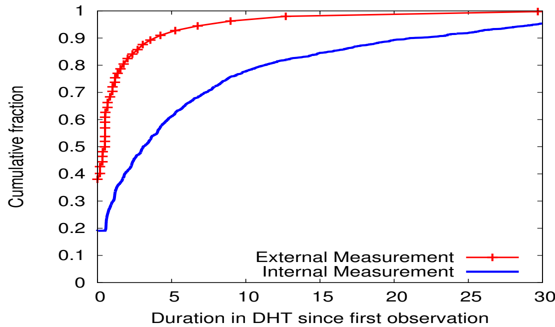


Figure 3: **Node Lifetimes in Vuze.**

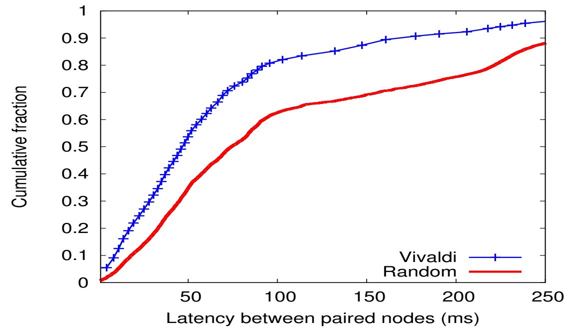


Figure 4: **Proximity of BitTorrent peers.**

measurement bias, but would require the measurement to be performed by nodes that are *within* the DHT.

Comet enables researchers to deploy experiments using measurement ASOs executed on nodes that are part of the DHT. To demonstrate the feasibility of this approach, we deployed Comet on 190 geographically dispersed PlanetLab nodes and integrated them into the production Vuze DHT. The measurement ASOs are stored on the Comet nodes, and they gather information about unmodified Vuze nodes that are adjacent (in the DHT) to the Comet nodes. We stored a lifetime measurement ASO (a variant of the code shown in Listing 6) at each of the Comet nodes, allowed the nodes to perform measurements for several days, and then collected and analyzed the data from these nodes.[5] Figure 3 plots the measurement data obtained from our experiments and compared to the lifetime data obtained by measurement nodes that are not integrated into the DHT (as in [20]). We observe that the measurements performed from within the DHT provide higher estimates for node lifetimes. The reason is that DHT-internal measurement nodes are able to traverse NATs in communicating with their neighbors. The difference is significant; we measured the median node lifetime as 3.1 hours, as opposed to an estimate of 0.5 hours obtained through conventional external measurements. Measurement ASOs are thus valuable tools in characterizing DHTs and provide more accurate data for tuning parameters such as replication factor, routing parallelism, etc.

6.3.2 Smart Rendezvous

In Section 5, we proposed a way to employ intelligent peer tracking for distributed P2P trackers using ASOs. We evaluate the usefulness of this application by deploying a distributed tracker built with Comet ASOs. As with traditional distributed trackers, clients participating in a P2P swarm (such as a BitTorrent download) register their participation by storing their IP addresses under the appropriate DHT key. In addition, clients also store their network coordinates (computed using Vivaldi [13]) along with their IP information. When clients contact the distributed tracker to obtain peer lists, the tracker ASO estimates the network latency between pairs of nodes using the supplied network coordinates and returns peers that are likely to be close to the requesting node. To evaluate this approach in practice, we deployed a tracker ASO on a Comet node in PlanetLab, while 190 PlanetLab nodes acted as peers in the swarm reporting their Vivaldi coordinates to the tracker and requesting good peers with which to communicate. Figure 4 depicts the effectiveness of this strategy compared to the default strategy of returning a random subset of peers to the requesting node. The graph shows a CDF of the measured latencies between peers under the two different matching schemes. The median value for the ASO-implemented Vivaldi intelligent peer matching is 47ms compared to a median of 72ms for the default scheme, a 35% latency improvement.

6.3.3 Vanish

Comet grew in part out of our experience specializing the Vuze DHT for Vanish [25], a self-destructing data system. Vanish used Vuze for key storage, however, Wolchock et al. [61] showed that the Vuze system was extremely open to a Sybil data harvesting attack that is able to scan the DHT for values. The attack worked in part because of Vuze's overly zealous replication policy – a high replication factor, coupled with a policy to replicate to new nodes immediately. In response, we set out to deploy new replication mechanisms and other anti-Sybil defenses in Vuze [24]. While these mechanisms were straightforward, deploying them required the cooperation of Vuze's designer and was an arduous and imperfect process. While many iterations would have been necessary to fully test and optimize policies, we often had only one shot to catch the two-month release cycle.[6]

[5] As Comet is not currently deployed by Vuze, the measurement ASOs are stored only on the nodes that we control. A more extensive deployment would allow us to obtain more samples quickly.

[6] It takes a week or more from release until 80% of the nodes in Vuze adopt changes. This is in addition to a typical release cycle Vuze

For the same reason we were unable to test individual changes in isolation as they had to be shipped in bundles in order to make progress in reasonable time.

We have used Comet to re-implement several of the changes that we deployed in Vuze. Those changes include the customizable replication scheme described in Section 5 (particularly a scheme that replicates only when the number of replicas falls below a threshold) and variable object lifetimes. As we showed in Section 5, both of these changes are trivial to program as Comet ASOs. Perhaps even more important, testing and re-deployment in Comet is significantly easier, as it does not require a redistribution of the entire DHT code base. Instead, new mechanisms can be deployed by overwriting the handler code for existing objects and using the updated bytecode for subsequently created objects, without requiring the involvement of the DHT administrators.[7] Had Comet existed at the time we deployed Vanish, it would have been possible to customize the DHT for the security requirements of the application from the start, and to optimize those policies to Vanish's requirements.

7 Security Analysis

The classic security goals for DHTs include resilience to attacks that: violate the system's ability to robustly store data [48], disrupt routing [48, 11], identify the participating nodes in the DHT [53, 51], and harvest copies of data stored within the DHT [61]. There are numerous well-known techniques aimed at violating these goals, including Sybil attacks [19], Eclipse attacks [47], and many others [55]. And there are also many known mechanisms for protecting against such attacks, including the use of strong identities minted by a logically centralized authority, computational puzzles and bandwidth contributions proofs [9, 16, 18, 63, 10], and architectures built upon social network structures [31, 63]. A production DHT with ASO support must consider such classic security goals, and can leverage known countermeasures for the corresponding threats. (Although, as exemplified by Vuze and other popular DHTs, a DHT for ASOs may decide that the risks associated with these threats are minimal, and hence not deploy the known defenses.)

The security concerns of DHTs with *signed* ASOs are roughly those of conventional DHTs without ASOs (since the signed ASOs can be viewed as "vetted" parts of the DHT system itself); we therefore do not consider signed ASOs further. Empowering DHTs with *unsigned* ASOs does, however, create a *new* potential attack vector not present in conventional DHTs – namely, attacks via malicious ASOs. We seek to ensure that a malicious ASO cannot: infer private information about or damage its Comet hosting node; infer information about or affect the properties of other ASOs stored within Comet; or infer private information about or affect the properties of other Comet nodes and arbitrary computers on the Internet. To place these goals in context, we stress that while an attacker could always use her own custom software to communicate with Comet in arbitrary ways, including putting to or getting from arbitrary ASO keys and communicating with the broader Internet in arbitrary ways, our goals – if attained – imply that ASOs cannot be used to amplify the attacker's resources or capabilities. For example, an attacker should not be able to create an ASO "worm" that spreads virally, mounting a DDoS attack against a victim ASO or device on the Internet.

We find that it is possible to meet these goals using three architectural features: (1) restricting system access, (2) restricting resource consumption, and (3) restricting within-Comet communication. We consider each in turn.

Restricting system access. We designed the ASO API to be highly restrictive. The API explicitly restricts an ASO's ability to infer private information about its host or to affect the host's state. The API similarly restricts an ASO's ability to interact with arbitrary devices on the Internet. For example, the API limits an ASO's IO capabilities to explicitly defined DHT operations; arbitrary disk, network, and other IO operations are prohibited. The API also prevents an ASO from introspecting its host; e.g., although we allow the ASO to learn its host's external IP, we explicitly prevent the ASO from learning its host's internal IP. Without these restrictions, an ASO could potentially read private files on the host's disk, write sensitive files, attempt to DoS an arbitrary remote node, map the network topology of internal IP networks, and so on. The Lua sandbox provides a simple mechanism for achieving this isolation. Namely, we removed the IO system call interface and exposed one containing only the restricted DHT operations instead.

Despite these restrictions, it may be possible for an ASO to infer (minimal) information about the hosting node via side-channels. For example, the time it takes an ASO to perform a computation could leak information to the ASO about the speed of the hosting processor. At the extreme, it may be feasible to infer modest information about other applications running on the hosting node [42]. We believe that such attacks are low risk in the Comet environment and do not consider them here.

Restricting resource consumption. Comet also significantly limits an ASO's ability to consume resources on its hosting node. Our prototype limits both the memory and CPU consumption of ASOs.

Memory. The Comet active runtime keeps a running

employs, which spans about a month.

[7] In general, updating the handler code for existing objects would require the application to keep track of its ASOs. In the case of applications such as Vanish, where objects are transient and have timeouts in the order of a few hours, we can also let existing objects just expire without explicitly updating them.

sum of the memory footprint of an ASO. Hard limits can be set on the total memory consumption of an object; ASOs which exceed this limit are evicted. Our current prototype limits ASOs to 100kB.

CPU. The Comet runtime similarly keeps a running count of bytecode operations performed. We envision multiple policies for constraining CPU use. The naive policy limits each ASO to at most a limited number of instructions per handler invocation. Since not all Lua operations are equally costly, a more sophisticated policy would assign different weights to different Lua operations (e.g., more cost for a table lookup than an addition). The limit could also be enforced over a fixed duration of time (such as 30 minutes) rather than upon each handler invocation (which might occur much more frequently). Our current prototype implements the naive restriction and allows 100K instructions per handler invocation.

Comet provides support for exception handling in order to help debug faulty ASOs that exceed the system-imposed resource limits. Handlers can catch resource exhaustion exceptions and store the relevant handler state as part of the ASO. The developer can then retrieve this stored state and inspect it to determine why the handler exceeded the resource limits. Further, operations that return values, e.g., `gets`, provide the stack trace as a return value in the case of an exception. We found these features to be useful in debugging many of the applications that we prototyped using Comet.

Restricting within-Comet communications. There are two classes of communications that we must consider: communications between one ASO and another, and callback communications to a caller.

Communications between ASOs. Allowing arbitrary between-ASO communications in Comet could lead to abuse. For example, suppose a malicious ASO stored under one key copies itself to a large number of other keys slowly over time, and then simultaneously all ASOs initiate connections to a victim ASO stored under some target key. Such an attack allows an attacker to amplify her resources: the attacker invests minimal effort to seed the original malicious ASO, yet the ultimate attack DDoSes nodes hosting the target key. Comet takes a Draconian approach toward protecting against such attacks: the ASO API only allows ASOs to communicate if they are stored under the *same* key, whether co-located on the same Comet node or on another node within the DHT. Our system further rate-limits communications performed by a particular ASO. Each Comet node allots a limited number of network communications per time period for every ASO it hosts. Though we have not experimentally ascertained appropriate rate-limiting parameters, the applications we present could all work with approximately the same number of network operations as is required for a value in the current Vuze DHT - about 20 every timer interval.

8 Conclusions

This paper described Comet, an active distributed key-value store. Comet enables clients to customize a distributed storage system in application-specific ways using Comet's active storage objects. By supporting ASOs, Comet allows multiple applications with diverse requirements to share a common storage system. We implemented Comet on the Vuze DHT using a severely restricted Lua language sandbox for handler programming. Our measurements and experience demonstrate that a broad range of behaviors and customizations are possible in a safe, but active, storage environment.

9 Acknowledgements

This work was supported in part by the National Science Foundation under grants NSF-0627367, NSF-0614975, NSF-0619836, NSF-0722004, and NSF-0963754, by the Google Fellowship in Cloud Computing, and by the Wissner-Slivka Chair. We thank Paul Gardner for his support on Vuze, and David Wetherall and our shepherd Wilson Hsieh for their helpful feedback on the paper.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evalaution. In *Proc. of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[2] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI*, 2002.

[3] Amazon S3. `http://aws.amazon.com/s3/`.

[4] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4), April 2005.

[5] Apache Cassandra. `http://cassandra.apache.org/`.

[6] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensible, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM Symp. on Operating systems Principles*, December 1995.

[7] B. N. Bershad and C. B. Pinkerton. Watchdogs – extending the UNIX file system. *Computer Systems*, 1(2), 1988.

[8] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proc. of SIGCOMM*, 2004.

[9] N. Borisov. Computational puzzles as Sybil defenses. In *Proc. of the Intl. Conference on Peer-to-Peer Computing*, 2006.

[10] N. Borisov. Computational puzzles as Sybil defenses. In *Proc. of the Intl. Conference on Peer-to-Peer Computing*, 2006.

[11] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 2002.

[12] D. R. Choffnes and F. E. Bustamante. Taming the Torrent: A practical approach to reducing cross-ISP traffic in P2P systems. In *Proc. of SIGCOMM*, 2008.

[13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. In *Proc. of SIGCOMM*, 2004.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, 2001.

[15] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *NSDI*, 2004.

[16] G. Danezis, C. Lesniewski-Laas, M. F. Kaashoek, and R. J. Anderson. Sybil-resistant DHT routing. In *ESORICS*, 2005.

[17] 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 *Proc. of SOSP*, 2007.

[18] J. Dinger and H. Hartenstein. Defending the Sybil Attack in P2P Networks: Taxonomy, Challenges, and a Proposal for Self-Registration. In *Intl. Conf. on Availability, Reliability and Security*, 2006.

[19] J. R. Douceur. The Sybil attack. In *Proc. of IPTPS*, 2002.

[20] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson. Profiling a million user DHT. In *Proc. of IMC*, 2007.

[21] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *NSDI*, pages 239–252, 2004.

[22] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *WORLDS'05*, pages 10–10, Berkeley, CA, USA, 2005. USENIX Association.

[23] P. Gardner. personal communication, 2009.

[24] R. Geambasu, T. Kohno, A. Krishnamurthy, A. Levy, H. M. Levy, P. Gardner, and V. Mascaritolo. Cascade: A compositional approach to self-destructing data. In Preparation, 2010.

[25] R. Geambasu, T. Kohno, A. Levy, and H. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of the USENIX Security Symposium*, August 2009.

[26] K. Hildrum and J. Kubiatowicz. Asymptotically Efficient Approaches to Fault-Tolerance in Peer-to-peer Networks. In *Proc. of International Symposium on Distributed Computing*, 2004.

[27] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Privacy-preserving P2P data sharing with OneSwarm. In *Proc. of SIGCOMM*, 2010.

[28] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, , and K. Mackenzie. Application performance and flexibility in exokernel systems. In *Proc. of SOSP*, 1997.

[29] K. Keetong, D. Patterson, and J. Hellerstein. A case for intelligent disks (IDISKs). *ACM SIGMOD Record*, 27(3), August 1998.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, December 1999.

[31] C. Lesniewski-Lass and M. F. Kaashoek. Whanaungatanga: Sybil-proof distributed hash table. In *Proc. of NSDI*, 2010.

[32] N. A. Lynch, D. Malkhi, and D. Ratajczak. Atomic Data Access in Distributed Hash Tables. In *Proc. of IPTPS*, 2001.

[33] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An Information Plane for Distributed Services. In *OSDI*, 2006.

[34] A. Muthitacharoen, S. Gilbert, and R. Morris. Etna: A fault-tolerant algorithm for atomic mutable DHT data. Technical Report MIT-LCS-TR-993, MIT, June 2005.

[35] Mysql Database Triggers. `http://dev.mysql.com/doc/refman/5.0/en/triggers.html`.

[36] L. Peterson, A. Bavier, M. Fiuczynski, and S. Muir. Experiences implementing PlanetLab. In *Proc. of OSDI*, 2006.

[37] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. Do incentives build robustness in BitTorrent? In *NSDI*, 2007.

[38] Project Voldemort. `http://project-voldemort.com/`.

[39] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. of NSDI*, Berkeley, CA, USA, 2004. USENIX Association.

[40] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. of SIGCOMM*, 2005.

[41] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. of 24th International Conference on Very Large Databases*, August 1998.

[42] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. of CCS*, 2009.

[43] W. C. F. Roberto Ierusalimschy, Luiz Henrique de Figueiredo. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1999.

[44] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of SOSP*, 2001.

[45] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proc. of the Third International COST264 Workshop on Networked Group Communication*, 2001.

[46] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI*, 1996.

[47] A. Singh, T.-W. Ngan, P. Druschel, , and D. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *INFOCOM*, 2006.

[48] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. of IPTPS*, 2002.

[49] M. Steiner and E. W. Biersack. Crawling AZUREUS. Technical report, Institut Eurecom, Networking and Security Department, 2008.

[50] M. Steiner, E. W. Biersack, and T. Ennajjary. Actively monitoring peers in KAD. In *Proc. of IPTPS*, 2007.

[51] M. Steiner, T. En-Najjary, and E. W. Biersack. A Global View of KAD. In *Proc. of IMC*, 2007.

[52] I. Stoica, D. Adkins, S. Zhuang, S. S. nker, and S. Surana. Internet indirection infrastructure. In *Proc. of SIGCOMM*, 2002.

[53] D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-Peer Networks. In *Proc. of IMC*, 2006.

[54] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *ACM SIGCOMM Computer Communications Review*, April 1996.

[55] G. Urdaneta, G. Pierre, and M. V. Steen. A Survey of DHT Security Techniques (to appear). *ACM Computing Survey*, 2010.

[56] uTorrent. `http://www.utorrent.com`.

[57] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of ISCA*, 1992.

[58] Vuze, Inc. `http://www.vuze.com`.

[59] P. Wang, I. Osipkov, N. Hopper, and Y. Kim. Myrmic: Secure and Robust DHT Routing. Technical report, University of Minnesota, 2007.

[60] D. Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *Proc. of the 17th ACM Symp. on Operating Systems Principles*, December 1999.

[61] S. Wolchok, O. S. Hofmann, E. W. Felten, J. A. Halderman, C. J. Rossbach, B. Waters, and E. Witchel. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *Proc. of NDSS*, 2010.

[62] H. Xie, R. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz. P4P: Provider portal for P2P applications. In *Proc. of SIGCOMM*, 2008.

[63] H. Yu, M. Kaminsky, P. B. Gibbons, and A. D. Flaxman. SybilGuard: defending against sybil attacks via social networks. *Proc. of SIGCOMM*, 2006.

[64] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of NOSSDAV*, 2001.