UNIVERSITY of WASHINGTON

# Pushing the Boundaries of Distributed Storage Systems


Hank Levy
Department of Computer Science & Engineering
University of Washington

with:
Roxana Geambasu (UW, Columbia University)
Amit Levy, Yoshi Kohno, Arvind Krishnamurthy (UW)

# Outline

- Introduction: Key/Value Stores, DHTs, etc.
- Vanish:  A Self-Destructing Data System
- Comet:  An Extensible Key/Value Store
- Conclusions/Summary

# Outline

- Introduction: Key/Value Stores, DHTs, etc.
- Vanish: A Self-Destructing Data System
- Comet: An Extensible Key/Value Store
- Conclusions/Summary

# Modern data and file sharing

- Over the last decade there has been a huge move to large-scale distributed data- and file-sharing systems
- Distributed Hash Tables (DHTs) have become a crucial mechanism for organizing scalable distributed Key/Value stores
- This move is impacting multiple environments: from mobile devices to desktops to global peer-to-peer file-sharing systems to data centers to cloud computing

# Intro to Distributed Hash Tables (DHTs)

- What's a Hash Table?
  - Data structure that maps "keys" to "values"
  - Extremely simple interface
    - put(key, value)
    - value = get(key)

- What's a Distributed Hash Table (DHT)?
  - Same thing, but the table is distributed across many hosts
  - There are tons of possible algorithms and protocols:
    - CAN, Chord, Pastry, Tapestry, Kademlia, Symphony, …

- Every node manages a contiguous segment of a huge (e.g., $2^{160}$) key space. Given a *key*, any node can *route* messages towards the node responsible for the key.

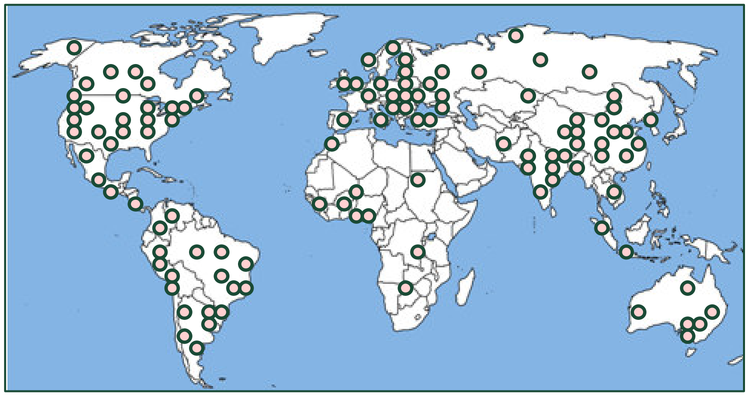
- This is managed at the *application* level.

# Why are DHTs interesting?

- Scalable
  - Highly robust to churn in nodes and data
  - Availability through data replication
- Efficient
  - Lookup takes *O(logN)* time
- Self-organizing and decentralized
  - No central point of control (or failure)
- Load balanced
  - All nodes are created equal (mostly)

# Intro to Peer-To-Peer (P2P) Systems

- A system composed of individually-owned computers that make a portion of their resources available directly to their peers *without* intermediary managed hosts or servers. [~wikipedia]



P2P properties:

- Huge scale – many millions of anonymous, autonomous nodes
- Geographic distribution – hundreds of countries
- Decentralization – individually-owned, no single point of trust
- Constant evolution – nodes constantly join and leave
- Examples – Kazaa, BitTorrent, Vuze, µTorrent, Napster, Skype, SETI@home

# DHTs and P2Ps

- DHTs provide a scalable, load-balanced, self-organizing structure
- DHTs are content addressable
  - Easy way for clients to share content and find content
    - E.g., key = hash("Lady Gaga"); data = get (key)
- Many P2P systems are therefore organized as DHTs
- There has been a lot of work at University of Washington on DHTs, including OneSwarm, BitTyrant, P4P, Vanish, Comet, ….
- In this talk I'm going to discuss two systems:
  - An (overly) challenging application (Vanish) [Usenix Security '09]
  - An extension of DHTs for the future (Comet) [OSDI '10]

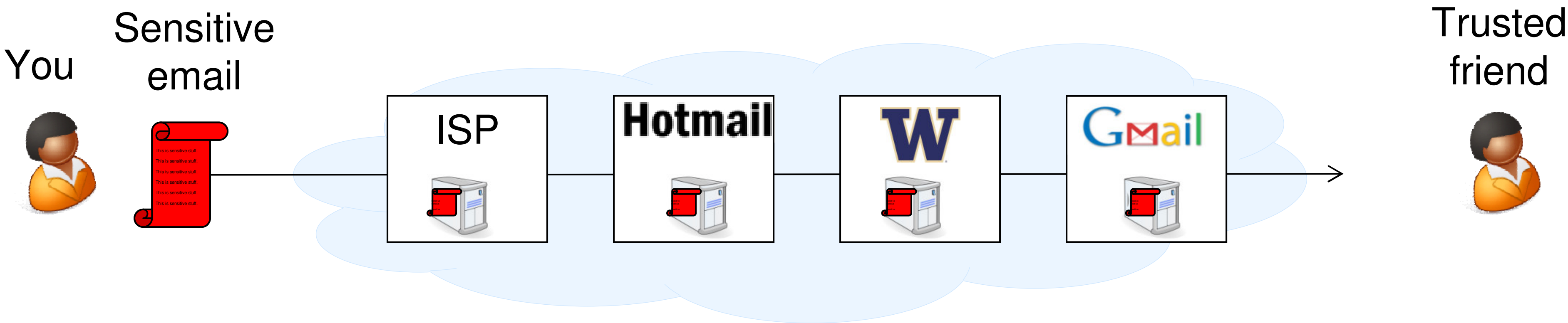
# Outline

- Introduction: Key/Value Stores, DHTs, etc.
- Vanish:  A Self-Destructing Data System
- Comet:  An Extensible Key/Value Store
- Conclusions/Summary

# The Problem: Data Lives Forever

- Huge disks have eliminated the need to ever delete data
  - Desktops store TBs of historical data
  - Phones, USB drives store GBs of personal data in your pocket
  - Data centers keep data forever
- The Web and cloud computing have made it impossible to delete data
  - Users have given up control of their data
  - Web services are highly replicated, archival stores
  - Data has value, services want to mine that value

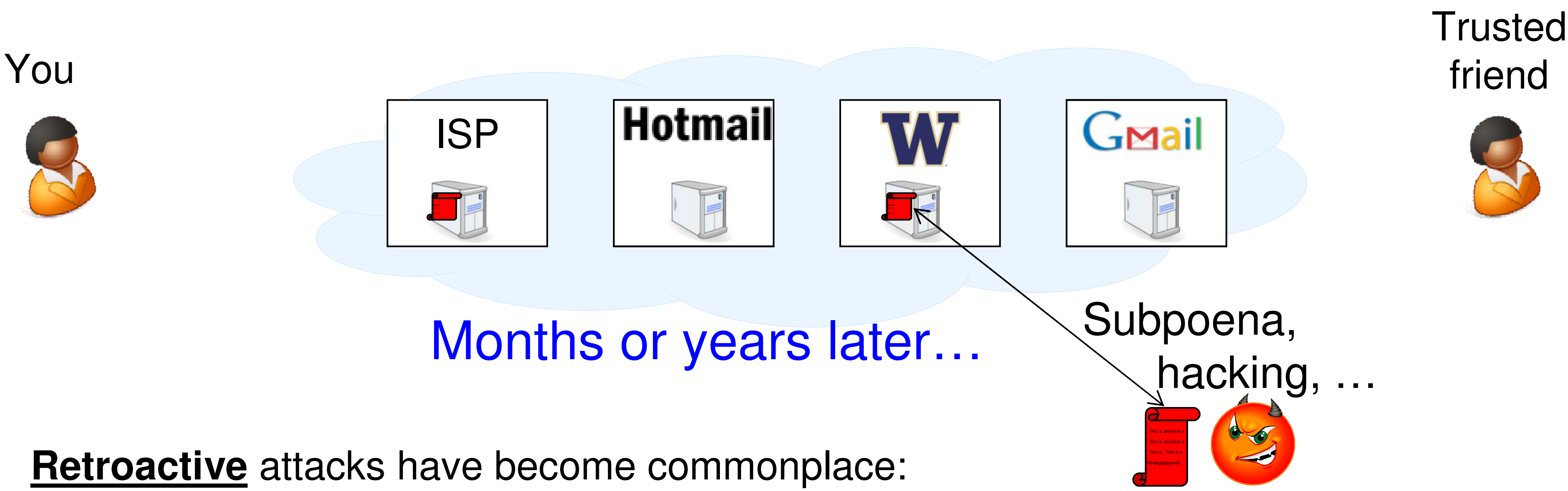
# Data Lives Forever: Example, Email




A few days later…

- You want to delete the email, but:
  - You don't know where all the copies are
  - You can't be sure that all services will delete all copies (e.g., from backups and replicas)
  - Even deleting your account doesn't necessarily delete the data (e.g., Facebook)

# Archived Copies Can Resurface Years Later




**Retroactive** attacks have become commonplace:

- Hackers
- Legal subpoena
- Misconfiguration
- Laptops seized a
- Device theft
- Carelessness
- …

Telegraph.co.

Chinese hum

WebProNews
Breaking eBusiness and Search News

Email Being Used More In Divorce Cases

say they have
g electronic data
ding to a survey
(AAML).

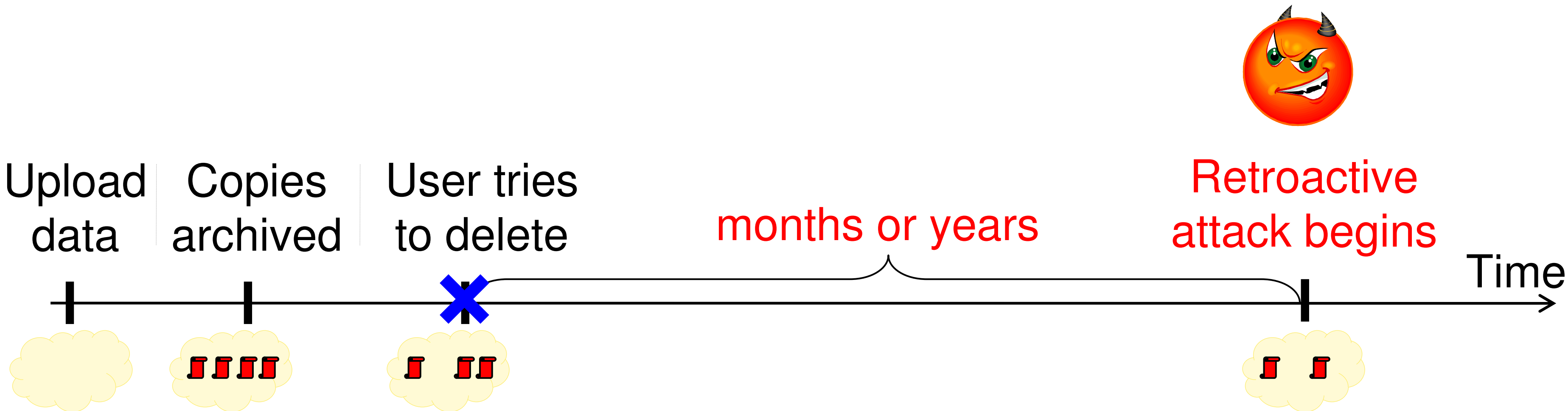
The New York Times

F.B.I. Gained Unauthorized Access to E-Mail

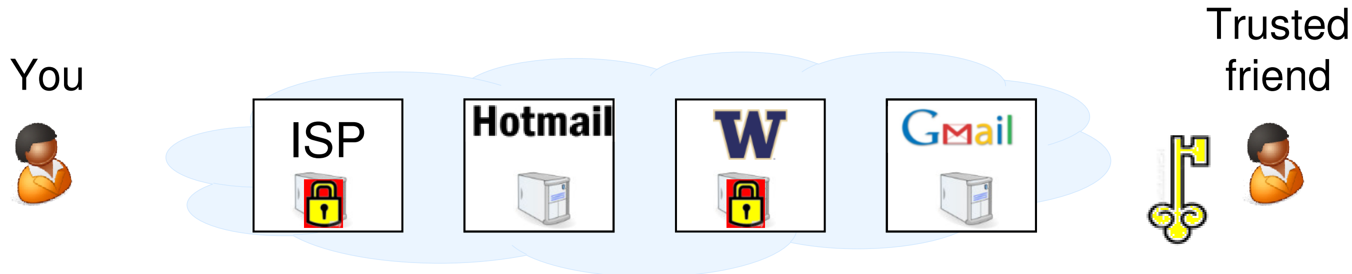
Published: February 17, 2008

WASHINGTON — A technical glitch gave the F.B.I. access to the e-mail messages from an entire computer network — perhaps hundreds of accounts or more — instead of simply the lone e-mail […]

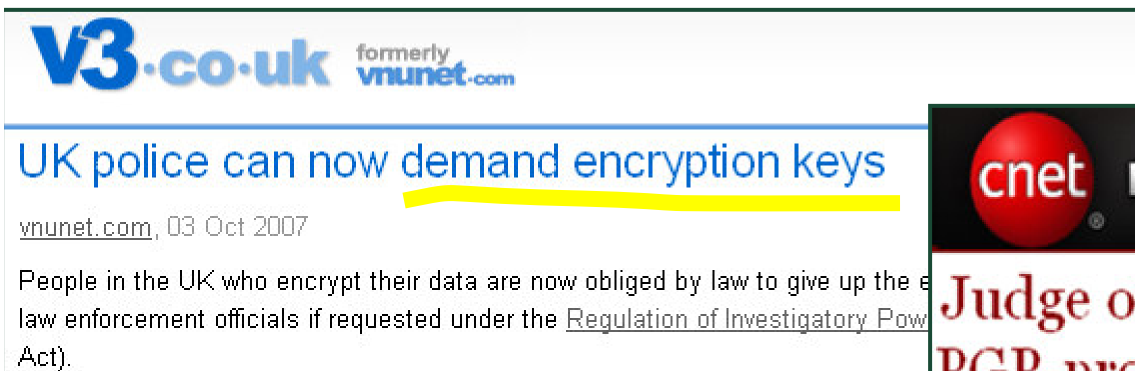
# The Retroactive Attack



# Why Not Rely On Encryption (e.g., PGP)?




- It's possible for an attacker to get both encrypted data and decryption key
  - PGP keys are long-lived (stored on disks, backed up)



V3.co.uk formerly vnunet.com

UK police can now demand encryption keys

vnunet.com, 03 Oct 2007

People in the UK who encrypt their data are now obliged by law to give up the e
law enforcement officials if requested under the Regulation of Investigatory Pow
Act).

cnet news

February 26, 2009 1:30 PM PST

Judge orders defendant to decrypt PGP-protected laptop

A federal judge has ordered a criminal defendant to decrypt his hard drive by typing in his PGP passphrase so prosecutors can view the unencrypted files, a ruling that raises serious concerns about self-incrimination in an electronic

# Why Not Rely On A Centralized Service?

**DeleteMyData.com**

Trust us: we'll help you delete your data!

- Huge trust concerns for relying on a centralized service

WIRED
BLOG NETWORK

**Encrypted E-Mail Company Hushmail Spills to Feds**

By Ryan Singel November 07, 2007 | 7:39:41 PM

Hushmail, a longtime provider of encrypted web-based email, markets itself by saying that "not even a Hushmail employee with access to our servers can read your encrypted e-mail, since each message is uniquely encoded before it leaves your computer."

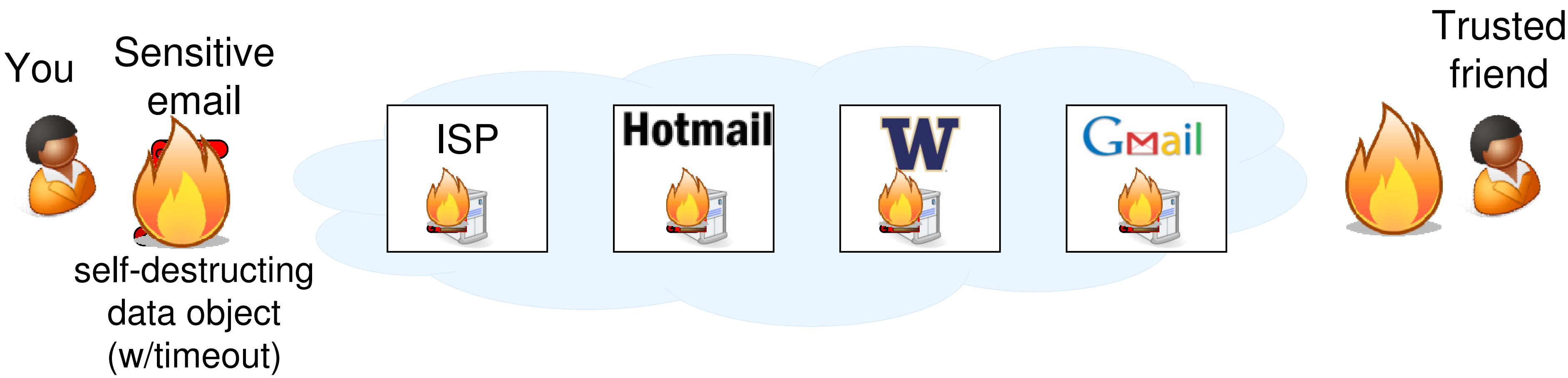
Question:

Can we empower users with control of data lifetime?

Answer:

Self-destructing data

# Self-Destructing Data Model

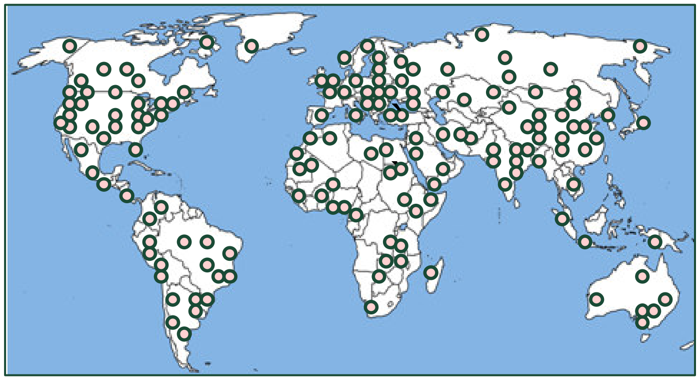



## Goals

1. Until timeout, users can read original data
2. After timeout, all copies become permanently unreadable
   - 2.1 both online and offline copies
   - 2.2 even for attackers who later obtain an archived copy & user keys
   - 2.3 without requiring any explicit action by the user
   - 2.4 without having to trust any centralized services

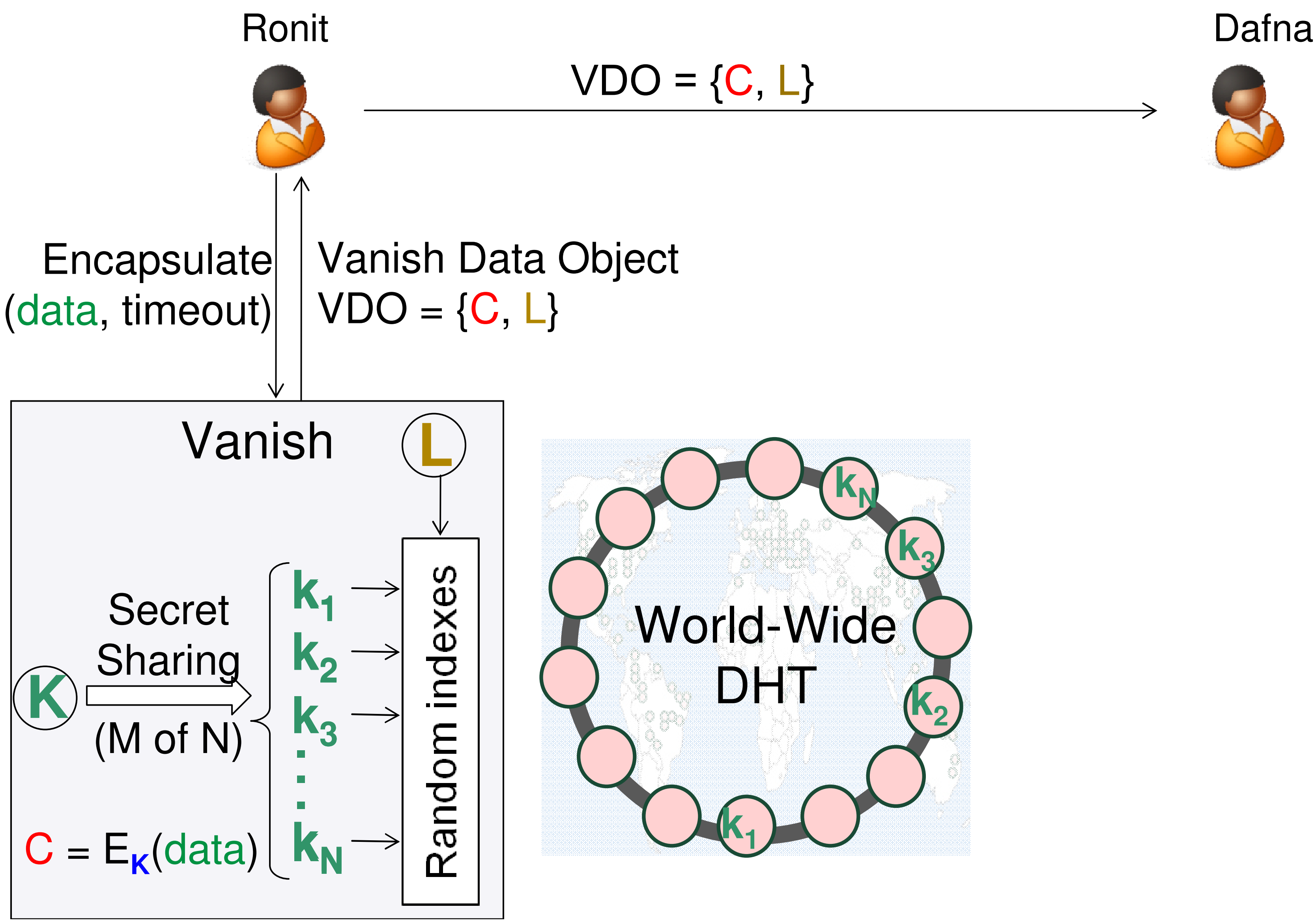
# One possibility: distributed trust systems

- Suppose we had access to millions of public “places” all around the world, where:
  - we could “hide” some information (*needle in a haystack*)
  - it would be impossible to find those locations later
  - the places would “lose” or time out our data over time (*churn*)
  - many independent trust domains

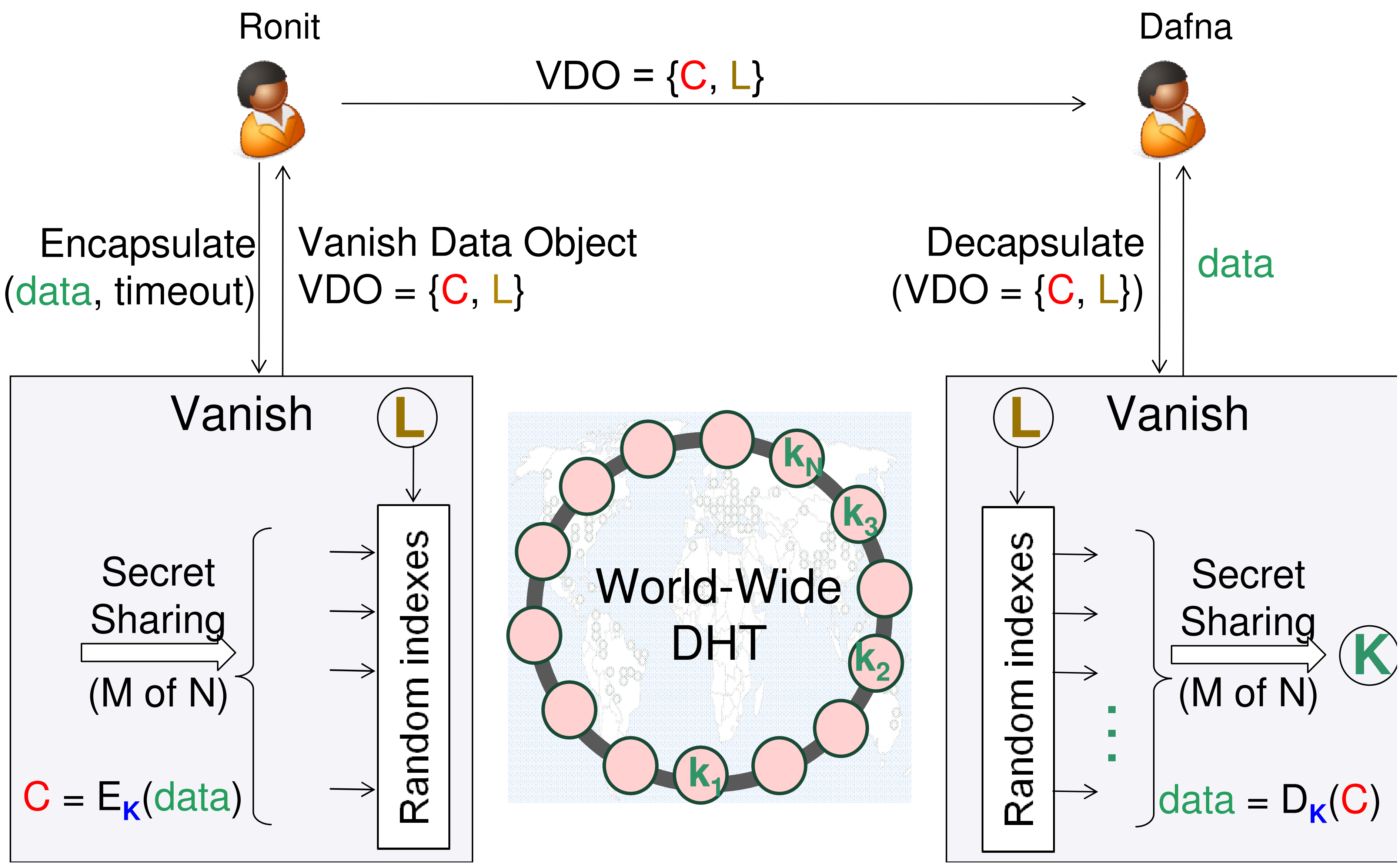


- How could we use this to create self-destructing data?

# One example: DHTs (Vanish)

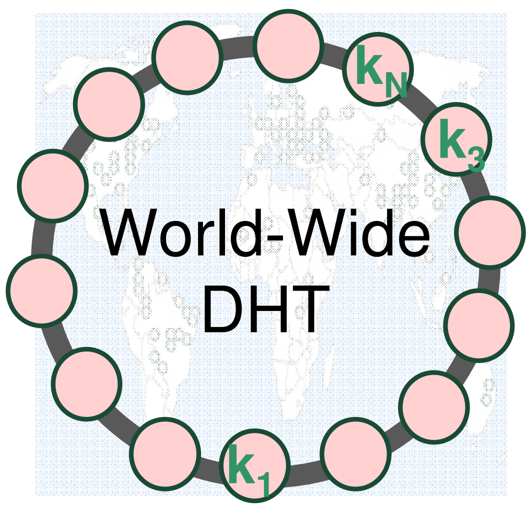


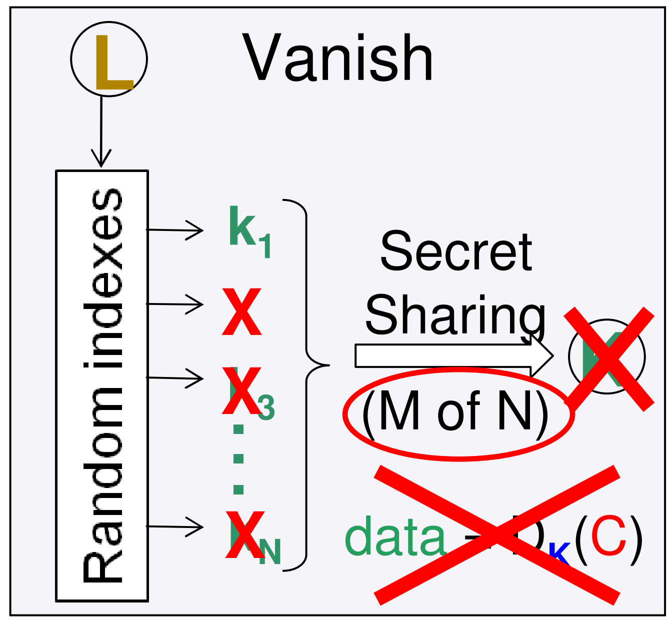
# How Vanish Works: Data Decapsulation



# How data times out in the DHT

- The DHT loses key pieces over time
  - Natural *churn*: nodes crash or leave the DHT
  - Built-in *timeout*: DHT nodes purge data periodically




- Key loss makes all data copies permanently unreadable
- Random indexes / node IDs are useless in the future

# Issues with DHTs for Vanish-like Apps

- We built the first Vanish prototype on Vuze – a commercial DHT with around 1.5M users [Geambasu et al. 2009].
- Vanish was really the first DHT application where security was a concern.
- After that time, it was shown that Vuze was open to data scanning attacks [Wolchok et al. 2010].
- We did a very detailed analysis of the threats and designed and deployed several changes to Vuze's commercial DHT.

# Security issues with Vuze and other DHTs

Vuze had two basic security issues:

1. Overly eager replication mechanisms
   - "push on join" sends copies of data to neighbors
   - aggressive 20-way replication every 30 minutes

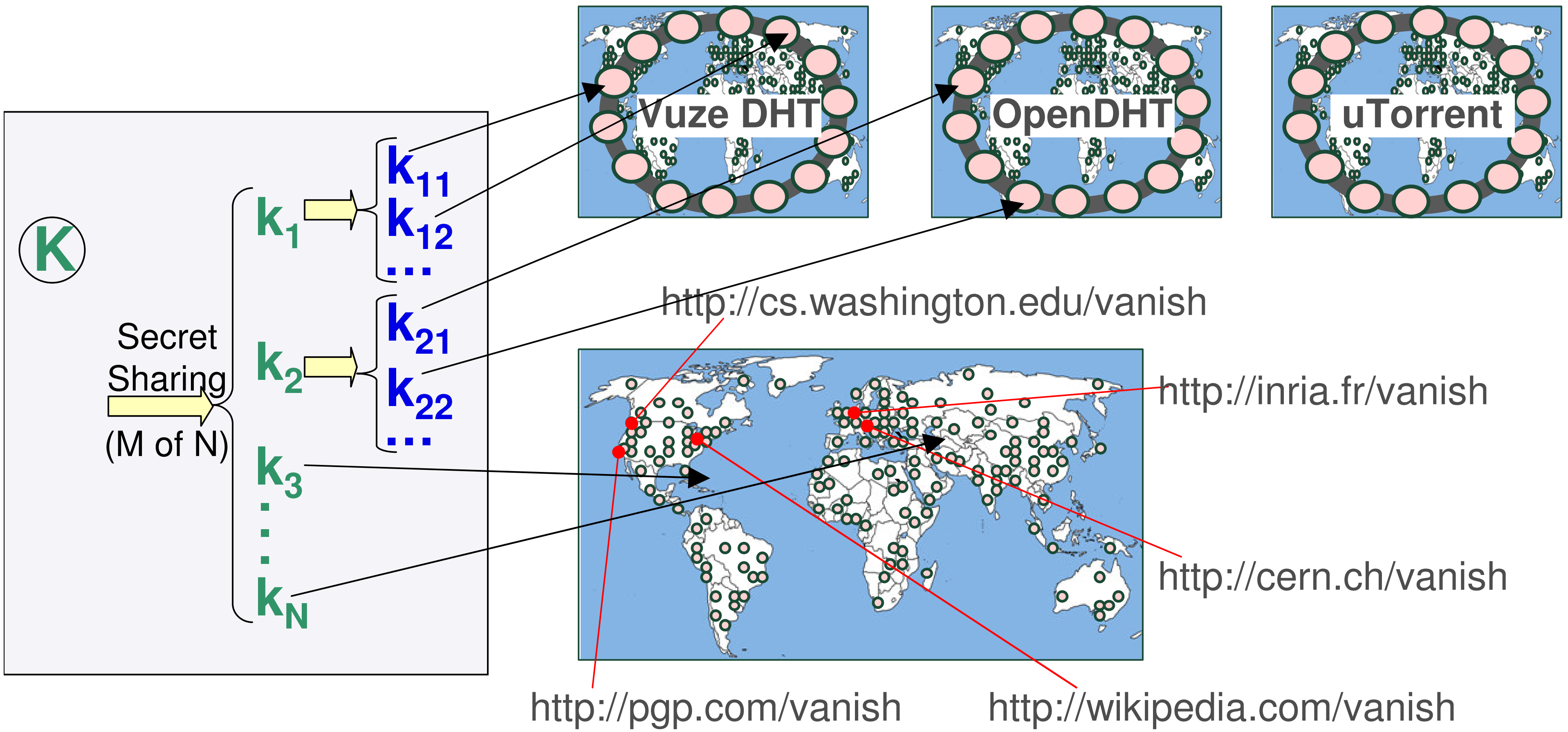
2. Lack of defense against Sybil attacks (where one node tries to join the DHT as many different clients)

# Changes to Vuze

- We designed and deployed many changes to Vuze for Vanish
  - Addition of explicit parameterized timeout
  - Removal of "push on join" replication
  - New "conditional replication" mechanism
    - Replicates only when needed
    - Replicates only as much as needed
  - New DHT ID assignment function
    - Acts as admission control mechanism
    - Limits number of clients joining from a single node
    - Limits number of clients joining from within a /16 network
    - Requires attacker to control a very diverse network (e.g., twenty /16 IP networks)
- Overall, raised the attack bar by many orders of magnitude

# Extending the trick:  hierarchical secret sharing

- Keys are spread over multiple *key/value storage systems*
- No single system has enough keys to decrypt the data



# Summary of self-deleting data

- Formidable challenges to privacy in the Web:
  - Data lives forever
  - Disclosures of data and keys have become commonplace
- Self-destructing data empowers users with lifetime control
- Our approach:
  - Combines secret sharing with global-scale, distributed-trust, decentralized systems to achieve data destruction
  - Can combine the best security properties of multiple systems to raise the bar against attack
  - Still lots of research to do here

# Summary (2)

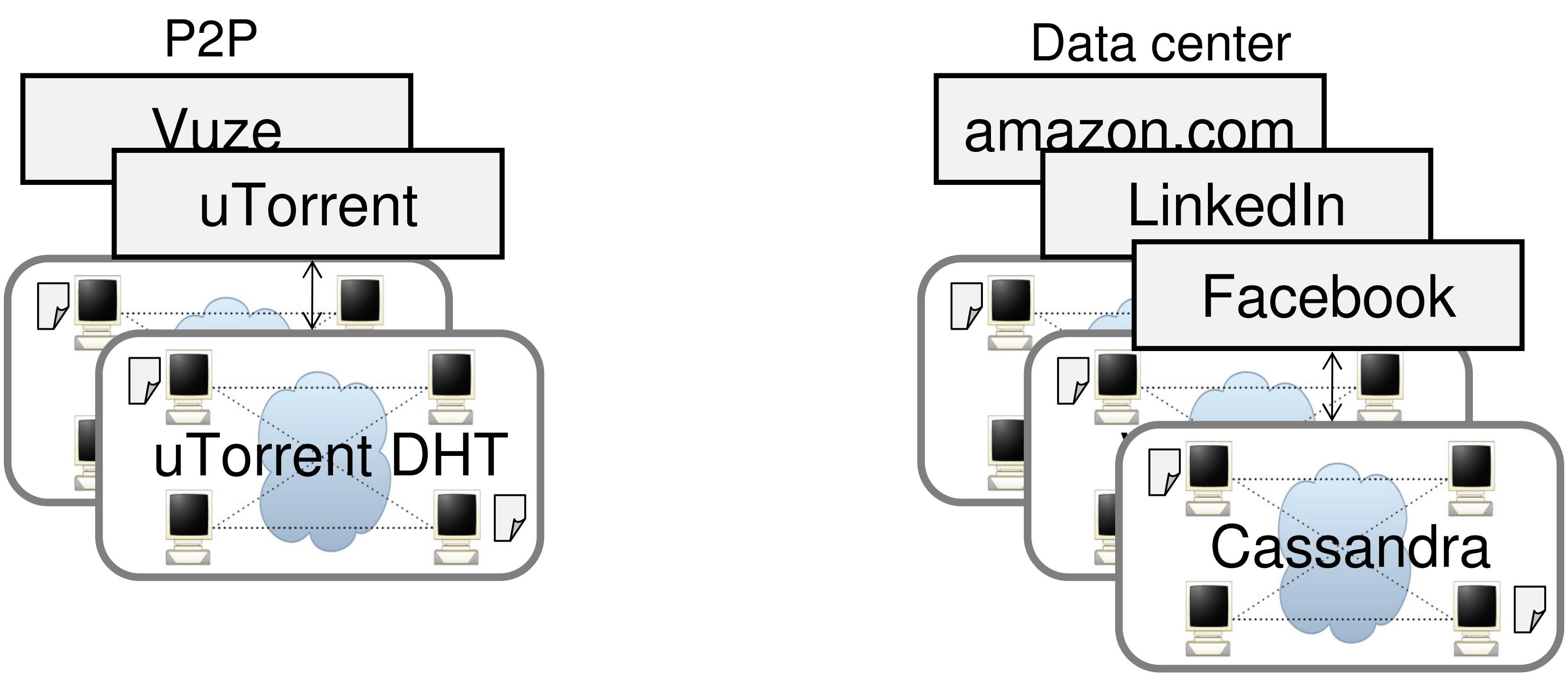
- This work stressed existing DHT designs
- It required us to design, measure, and deploy changes to a commercial, million-node, global-scale distributed key/value store.
- The changes were conceptually simple; but deploying them in a real system was hard.
- Question: can we make life easier for the next person who walks this path?

# Outline

- Introduction: Key/Value Stores, DHTs, etc.
- Vanish: A Self-Destructing Data System
- Comet: An Extensible Key/Value Store
- Conclusions/Summary

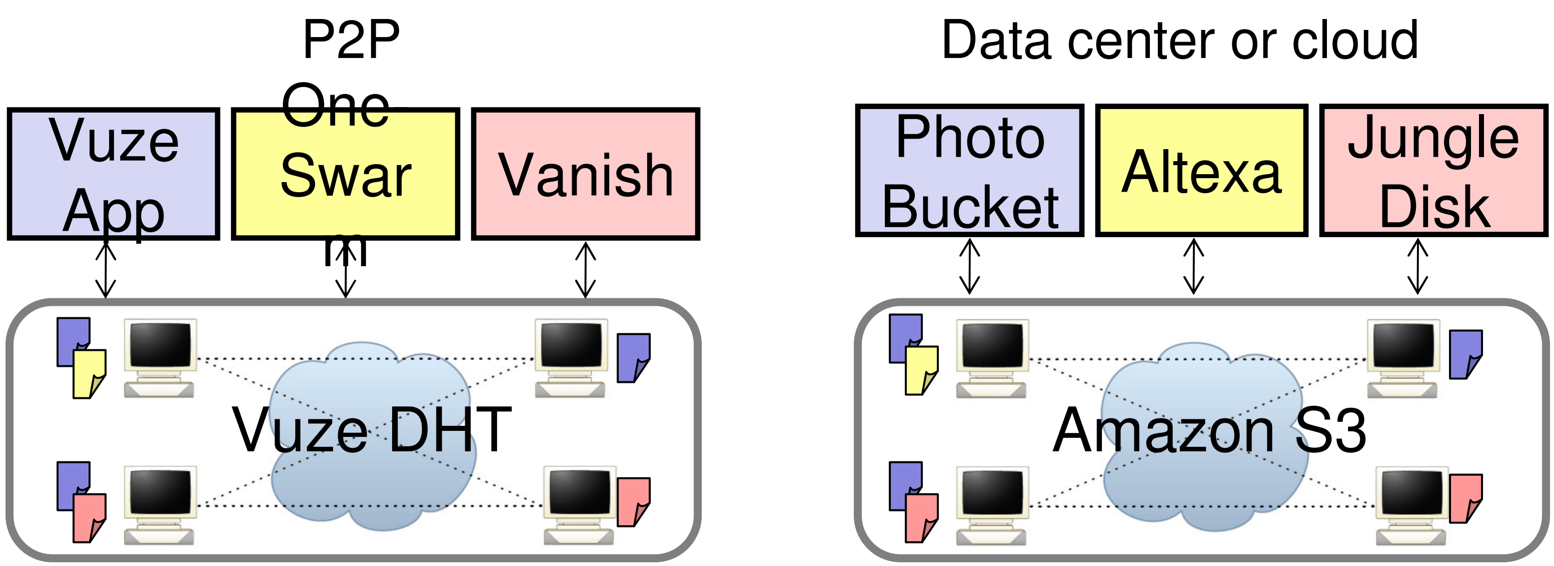
# Use of Distributed Key/Value Stores

- Key/Value stores are increasingly popular *both* in P2P systems and within data centers, for many reasons, e.g.: scalability, availability, load balancing, etc.



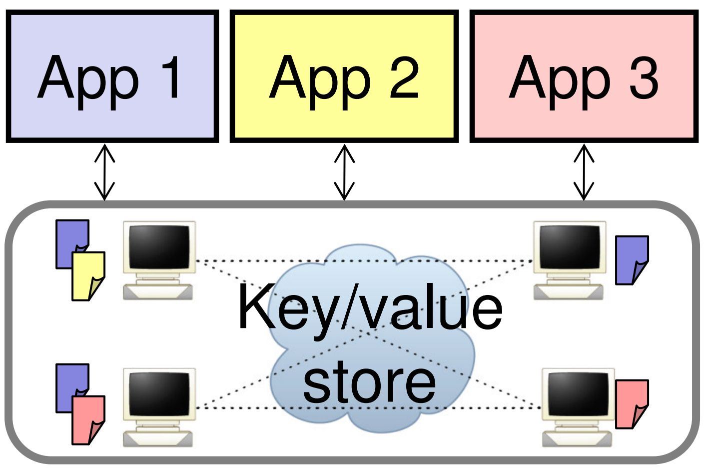
# Shared Key/Value Storage Services

- Increasingly, key/value stores are shared by many apps
  - Avoids per-app storage system deployment
- However, building apps atop today's stores is challenging



# Challenge: Inflexible Key/Value Stores

- Applications have different (even conflicting) needs:
  - Availability, security, performance, functionality
- But today's key/value stores are one-size-fits-all

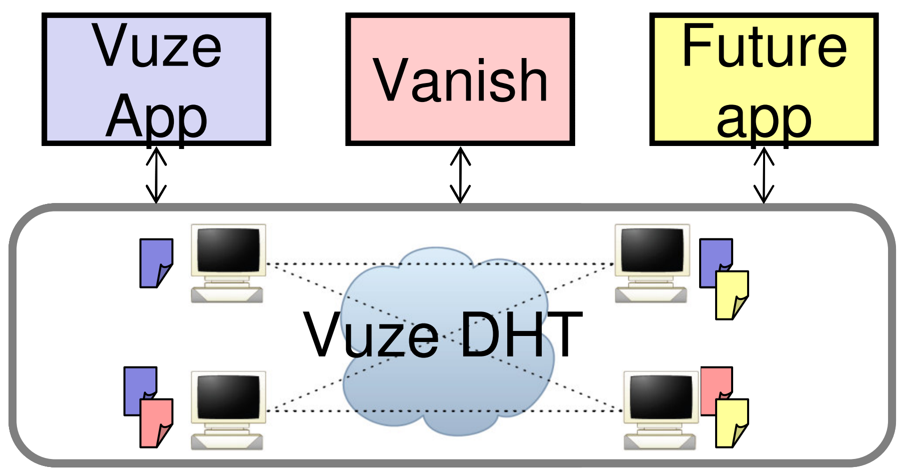


# Vanish was a motivating example

- Vuze design caused problems for Vanish:
  - Fixed 8-hour data timeout
  - Overly ag
- Changes we
  - Needed t
  - Long depl
  - Hard to evaluate before deployment

**Question:**

**How can a key/value store support many applications with different needs?**



# Extensible (*Active*) Key/Value Stores

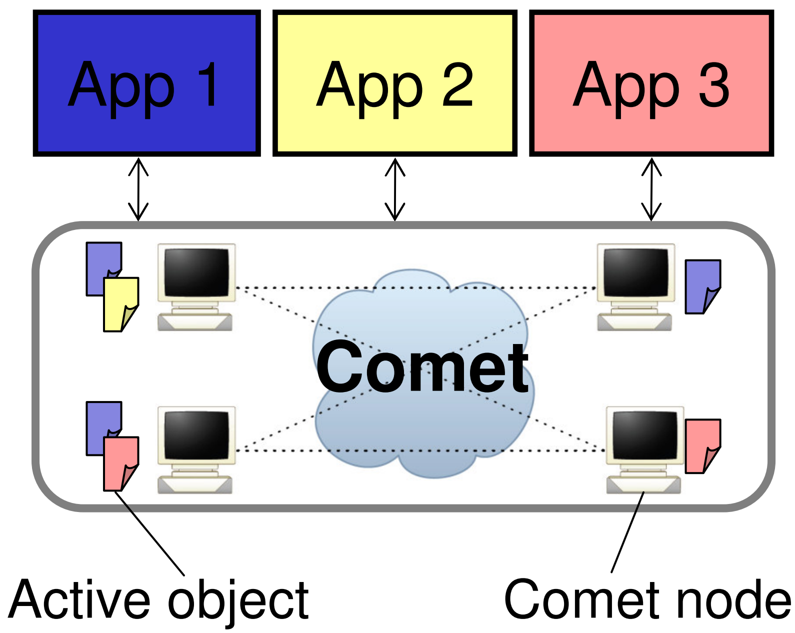
- Allow apps to customize the store's functions
  - Different data lifetimes
  - Different numbers of replicas
  - Different replication intervals
- Allow apps to define new functions
  - Tracking popularity: data item counts the number of reads
  - Access logging: data item logs readers' IPs
  - Adapting to context: data item returns different values to different requestors

# Comet Design Philosophy

- We want an extensible key/value store
- But we want to keep it simple!
  - Allow apps to inject tiny code fragments (10s of lines of code)
  - Adding even a tiny amount of programmability into key/value stores can be extremely powerful
  - We are *not* trying to be fully general
- Our Comet paper [OSDI ‘10] shows how to build extensible P2P DHTs
  - We leverage our DHT experience to drive our design

# Comet

- DHT that supports application-specific customizations
- Applications store active objects instead of passive values
  - Active objects contain small code snippets that control their behavior in the DHT

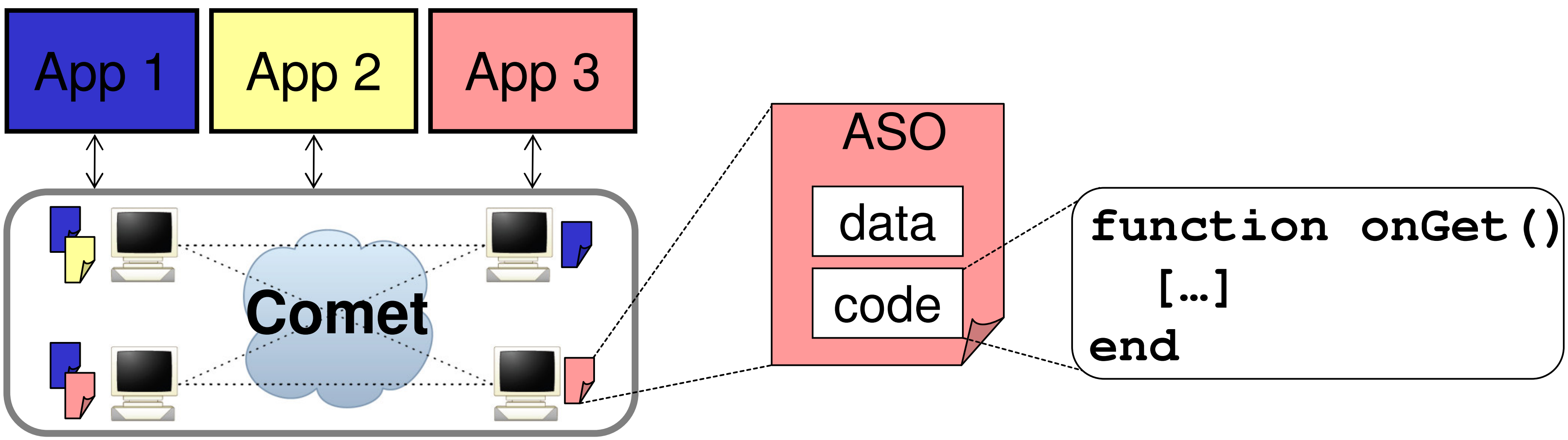


# Comet’s Goals

- Flexibility
  - Support a wide variety of small, lightweight customizations
- Isolation and safety
  - Limited knowledge, resource consumption, communication
- Lightweight
  - Low overhead for hosting nodes

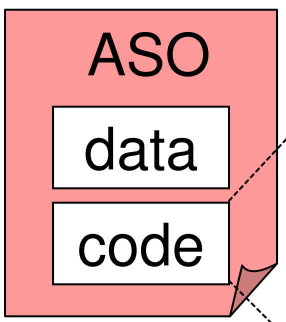
# Active Storage Objects (ASOs)

- Instead of storing <key,value>, Comet stores <key, ASO>
- The ASO consists of data and code
  - The data is the value
  - The code is a set of handlers that are called on **`put`**/**`get`**



# Simple ASO Example

- Each replica keeps track of number of **gets** on an object

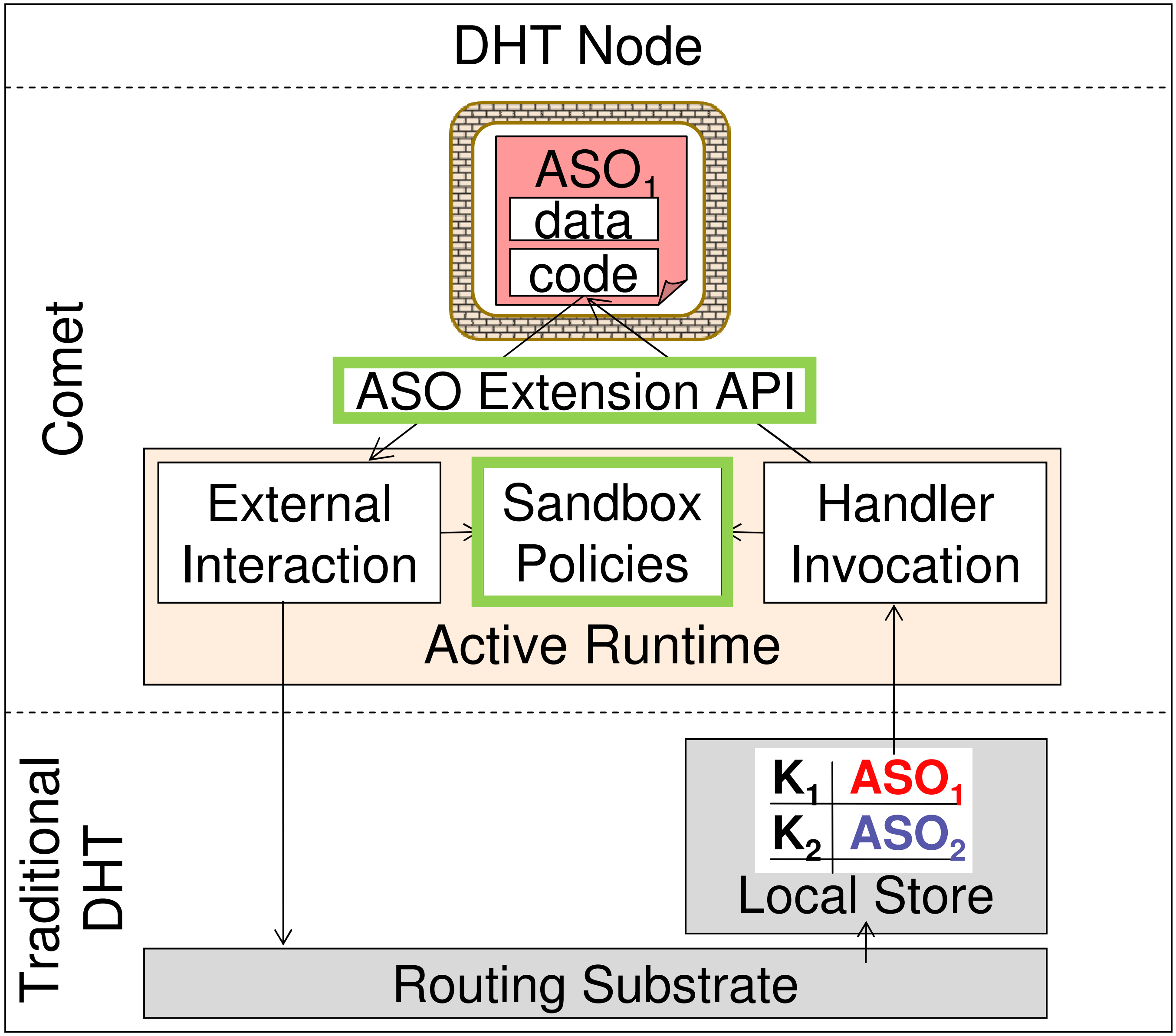



```
aso.value = "Hello Haifa!"
aso.getCount = 0

function onGet()
   self.getCount = self.getCount + 1
   return {self.value, self.getCount}
end
```

- The effect is powerful:
  - Difficult to track object popularity in today's DHTs
  - Trivial to do so in Comet without DHT modifications

# Comet Architecture



# The ASO Extension API

| Applications | Customizations |
|---|---|
| Vanish | Replication |
| | Timeout |
| | One-time values |
| Adeona | Password access |
| | Access logging |
| P2P File Sharing | Smart tracker |
| | Recursive gets |
| P2P Twitter | Publish / subscribe |
| | Hierarchical pub/sub |
| Measurement | Node lifetimes |
| | Replica monitoring |

# The ASO Extension API

| Intercept accesses | Periodic Tasks | Host Interaction | DHT Interaction |
|---|---|---|---|
| **`onPut`**(*caller*) | **`onTimer`**() | **`getSystemTime`**() | **`get`**(*key, nodes*) |
| **`onGet`**(*caller*) | | **`getNodeIP`**() | **`put`**(*key*, *data*, *nodes*) |
| **`onUpdate`**(*caller*) | | **`getNodeID`**() | **`lookup`**(*key*) |
| | | **`getASOKey`**() | |
| | | **`deleteSelf`**() | |

- Small yet powerful API for a wide variety of applications
  - We built over a dozen application customizations
- We have explicitly chosen not to support:
  - Sending arbitrary messages on the Internet
  - Doing I/O operations
  - Customizing routing …

# Restricting Active Storage Objects

- We restrict ASOs in three ways:
  - Limited knowledge
  - Limited resources
  - Limited DHT interaction

# Limited Knowledge

Requirement:

- It should be impossible for an ASO to access, e.g.:
  - User's files, local services, other ASOs on the node

Solution:

- We use a standard language sandbox
- ASOs are coded in a limited and lightweight language
  - The basis is Lua, a popular language for application extensions
    - Used for extending SimCity, World of Warcraft, Photoshop, …
  - We modify Lua to eliminate any unneeded functions:
    - E.g.: no process/thread creation, file I/O, sockets, signals, …
- ASO runtime is tiny (< 5,000 LOC)
  - Could be even model-checked

# Limited Resource Consumption

Requirement:

- Limit resource consumption by each ASO and by Comet
  - CPU, memory, bandwidth

Solution:

- We modified the Lua interpreter to limit:
  - Per-handler Lua bytecode instructions
  - Per-ASO and per-handler memory allocation
- We rate-limit incoming and outgoing ASO requests
- We limit the number of ASOs stored on each node

# Limited DHT Interaction

Requirement:

- The DHT-interaction API must not be exploitable
  - E.g.: prevent DDoS attacks against DHT nodes

Solution:

- Restrict who ASOs can talk to:
  - ASOs can initiate interactions only with their own neighbors
  - ASOs cannot send arbitrary network packets

# Comet Prototype

- We built Comet on top of Vuze and Lua
  - We deployed experimental nodes on PlanetLab
- In the future, we hope to deploy at a large scale
  - Vuze engineer is particularly interested in Comet for debugging and experimentation purposes

# Comet Applications

| Applications | Customization | Lines of Code |
|---|---|---|
| Vanish | Security-enhanced replication | 41 |
| | Flexible timeout | 15 |
| | One-time values | 15 |
| Adeona | Password-based access | 11 |
| | Access logging | 22 |
| P2P File Sharing | Smart Bittorrent tracker | 43 |
| | Recursive gets* | 9 |
| P2P Twitter | Publish/subscribe | 14 |
| | Hierarchical pub/sub* | 20 |
| Measurement | DHT-internal node lifetimes | 41 |
| | Replica monitoring | 21 |

* Require signed ASOs (see paper)

# Three Examples

1. Application-specific DHT customization
2. Context-aware storage object
3. Self-monitoring DHT

# 1. Application-Specific DHT Customization

- Example: customize the replication scheme

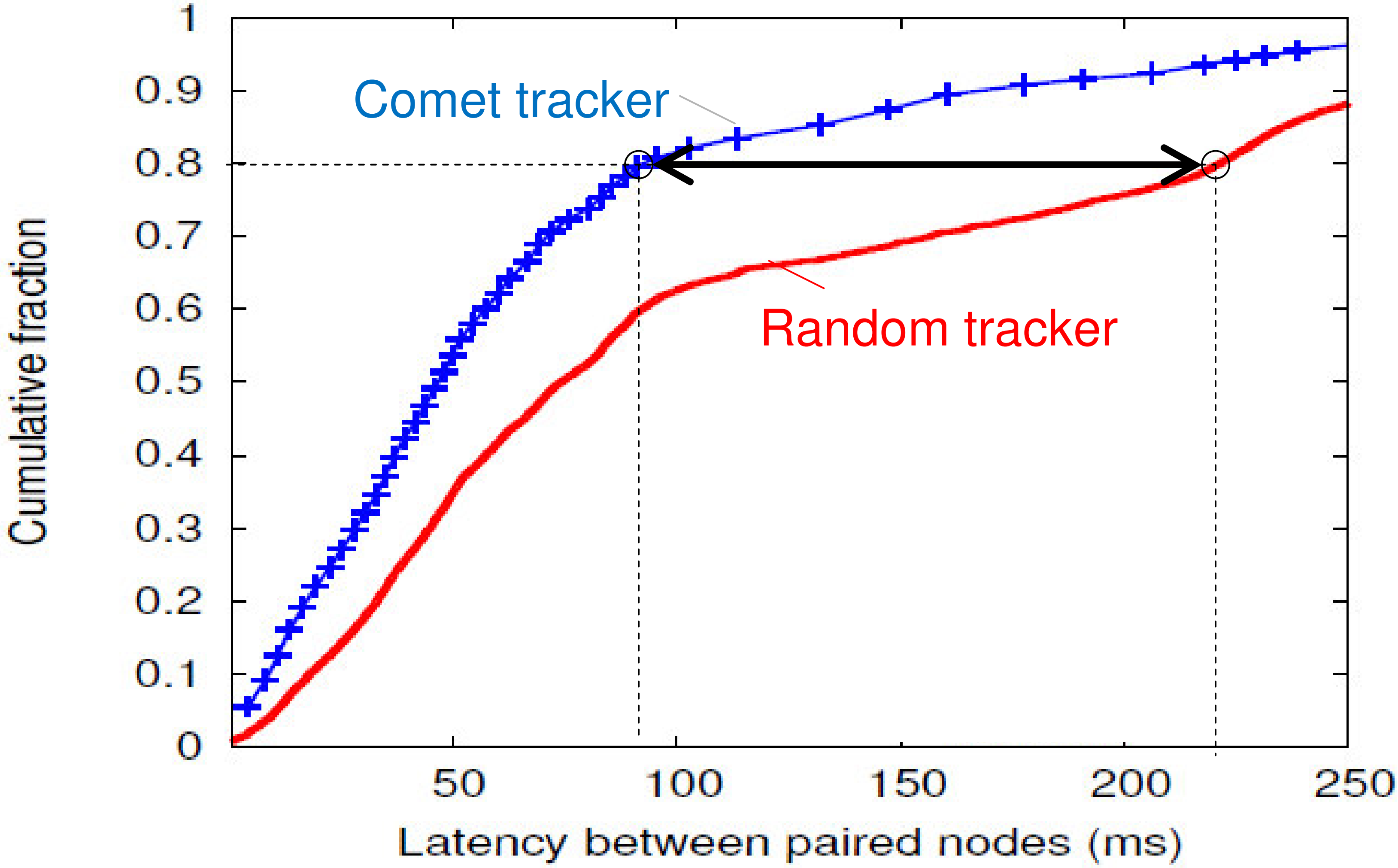
```
function aso:selectReplicas(neighbors)
  [...]
end
function aso:onTimer()
  neighbors = comet.lookup()
  replicas = self.selectReplicas(neighbors)
  comet.put(self, replicas)
end
```

- We have implemented the Vanish-specific replication
  - Code is 41 lines in Lua

# 2. Context-Aware Storage Object

- Traditional distributed trackers return a randomized subset of the nodes
- Comet: a proximity-based distributed tracker
  - Peers **`put`** their IPs and Vivaldi coordinates at **`torrentID`**
  - On **`get`**, the ASO computes and returns the set of closest peers to the requestor
- ASO has 37 lines of Lua code

# Proximity-Based Distributed Tracker



# 3. Self-Monitoring DHT

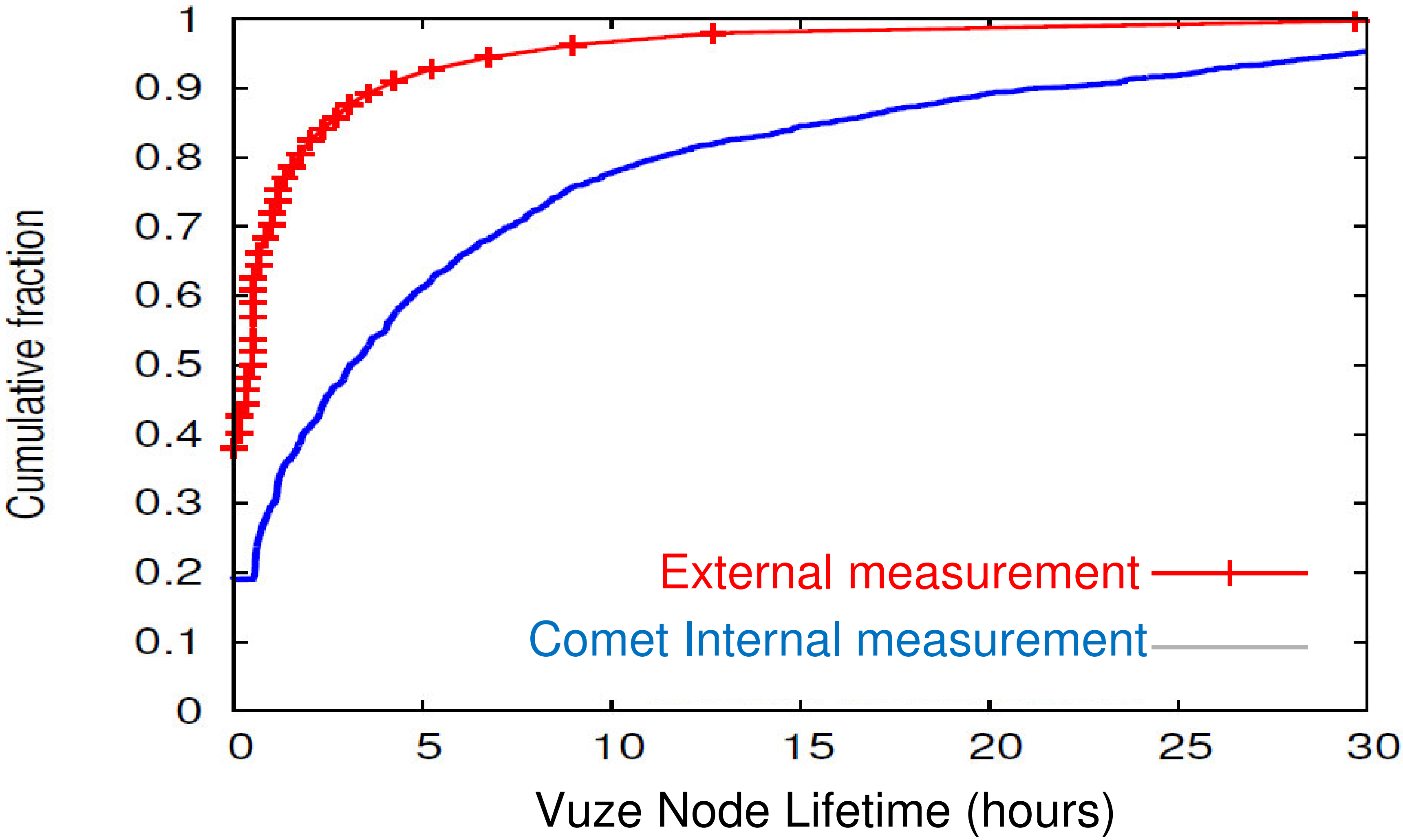
- Example: monitor a remote node's neighbors
  - **Put** a monitoring ASO that "pings" its neighbors periodically

```
aso.neighbors = {}

function aso:onTimer()
  neighbors = comet.lookup()
  self.neighbors[comet.systemTime()] = neighbors
end
```

- Useful for internal measurements of DHTs
  - Provides additional visibility over external measurement (e.g., NAT/firewall traversal)

# Example Measurement: Vuze Node Lifetimes



# Comet Summary

- Extensibility allows a shared storage system to support applications with different needs
- Comet is an extensible DHT that allows per-application customizations
  - Limited interfaces, language sandboxing, and resource and communication limits
  - Opens DHTs and key/value stores to a new set of stronger applications
- Extensibility is likely useful in data centers (e.g., S3):
  - Assured delete
  - Logging and forensics
  - Storage location awareness
  - Popularity

# Outline

- Introduction: Key/Value Stores, DHTs, etc.
- Vanish: A Self-Destructing Data System
- Comet: An Extensible Key/Value Store
- Conclusions/Summary

# Conclusions

- There will be a lot of key/value stores in our future
- There will be new applications generating new requirements
- These applications will be *sharing* a small number of key/value storage services (e.g., in data centers or the cloud)

- A small amount of programmability can:
  1. greatly increase the generality and usability of simple key/value stores, and
  2. facilitate new classes of applications.

# Questions?

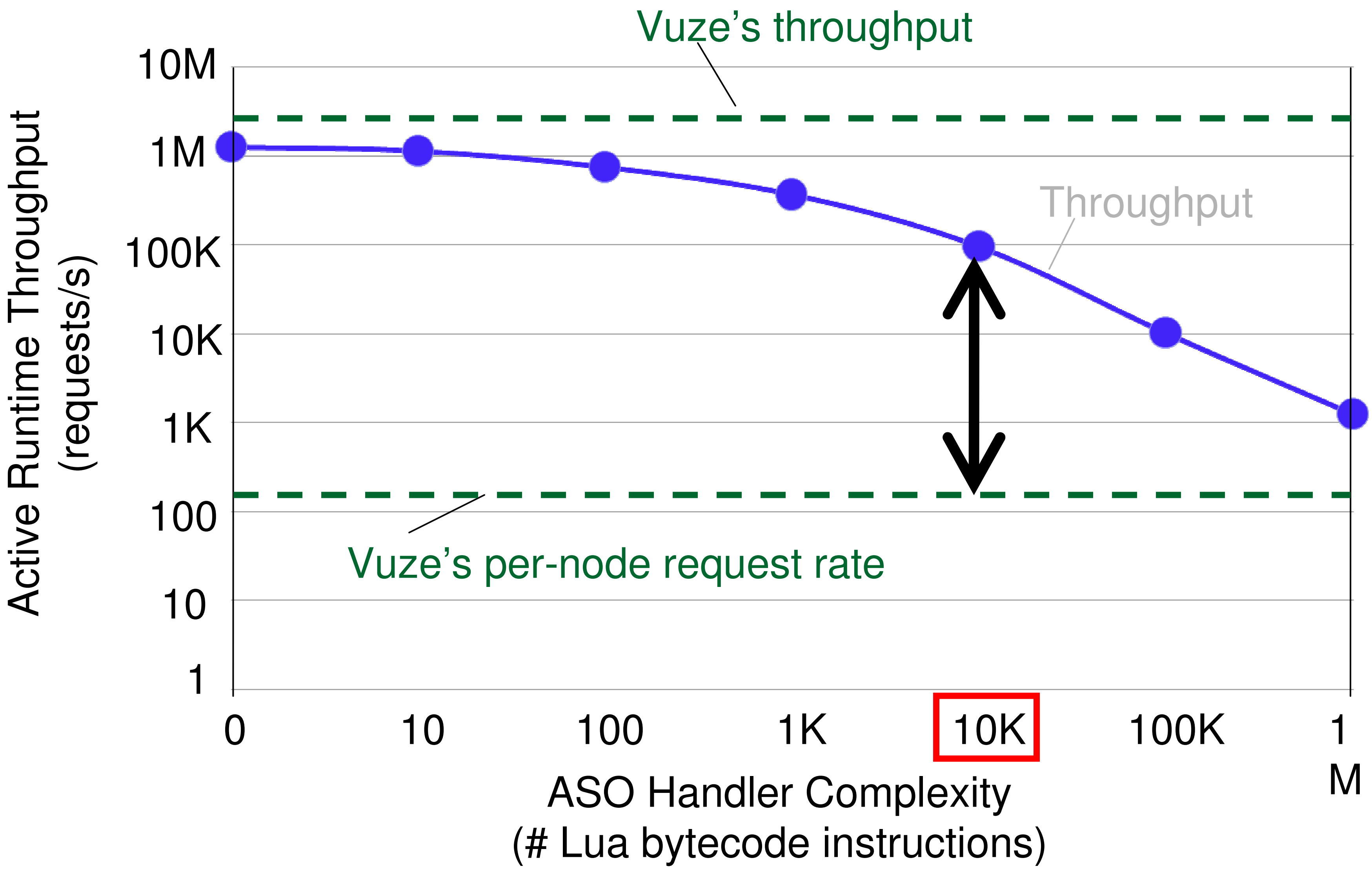
- Thanks to:
  - Roxana Geambasu, Amit Levy, Yoshi Kohno, Arvind Krishnamurthy (UW)
  - Paul Gardner (Vuze Inc.)

# Appendix

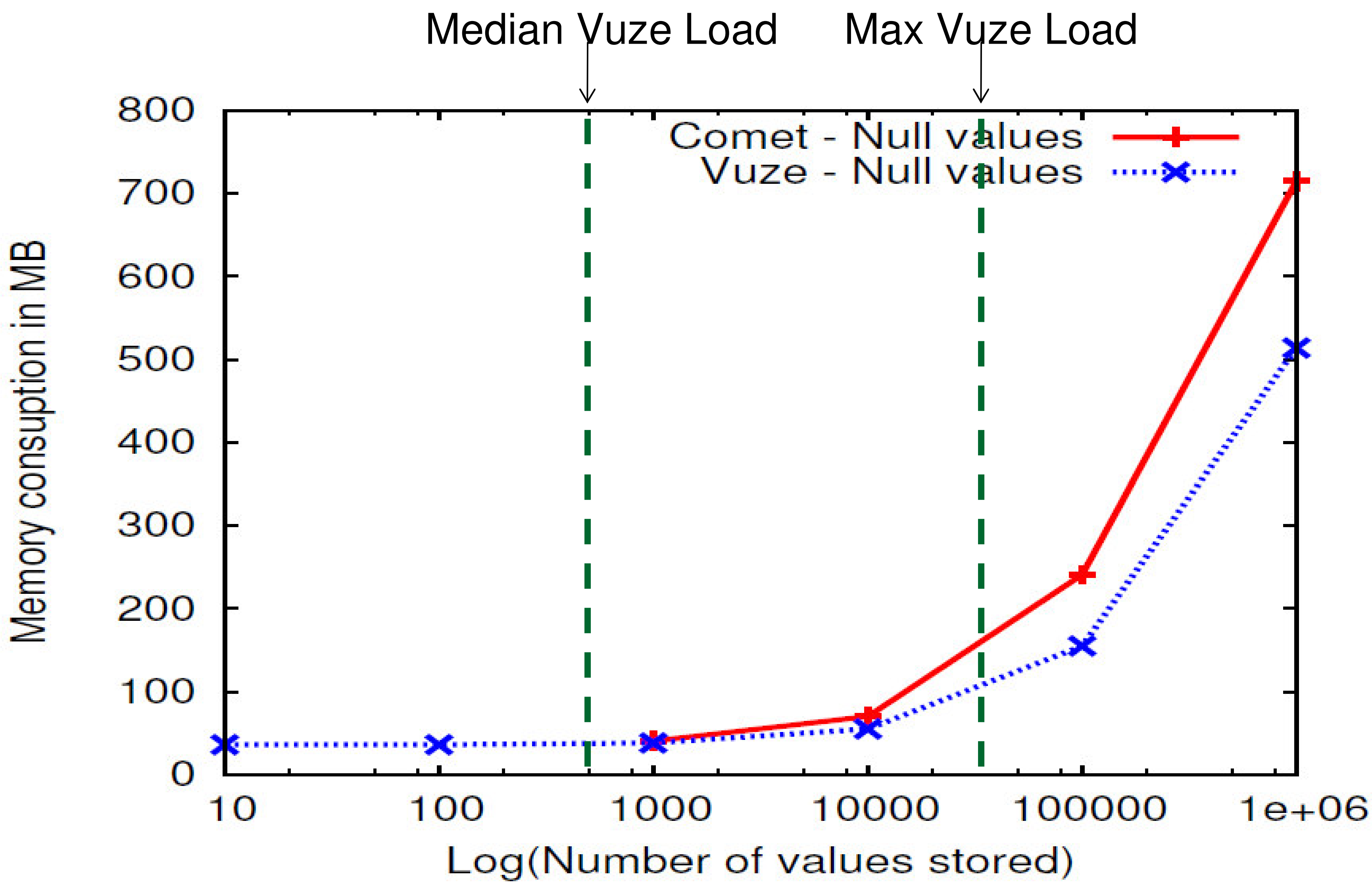
# Expected App. Resource Consumption

| 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 | $\approx 10$ | $4\mu s$ | $0.434K$ | $< 1K$ |
| Limited-Read Value | $\approx 10$ | $4\mu s$ | $0.553K$ | $< 1K$ |
| 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 | $\approx 10$ | $4\mu s$ | $0.971K$ | $3K$/hr |
| Smart Rendezvous | $1,000$s | $14\mu s$ | $1.107K$ | $10K$ |
| Recursive Get | $\approx 50$ | $6\mu s$ | $0.714K$ | $\approx 1K$ |

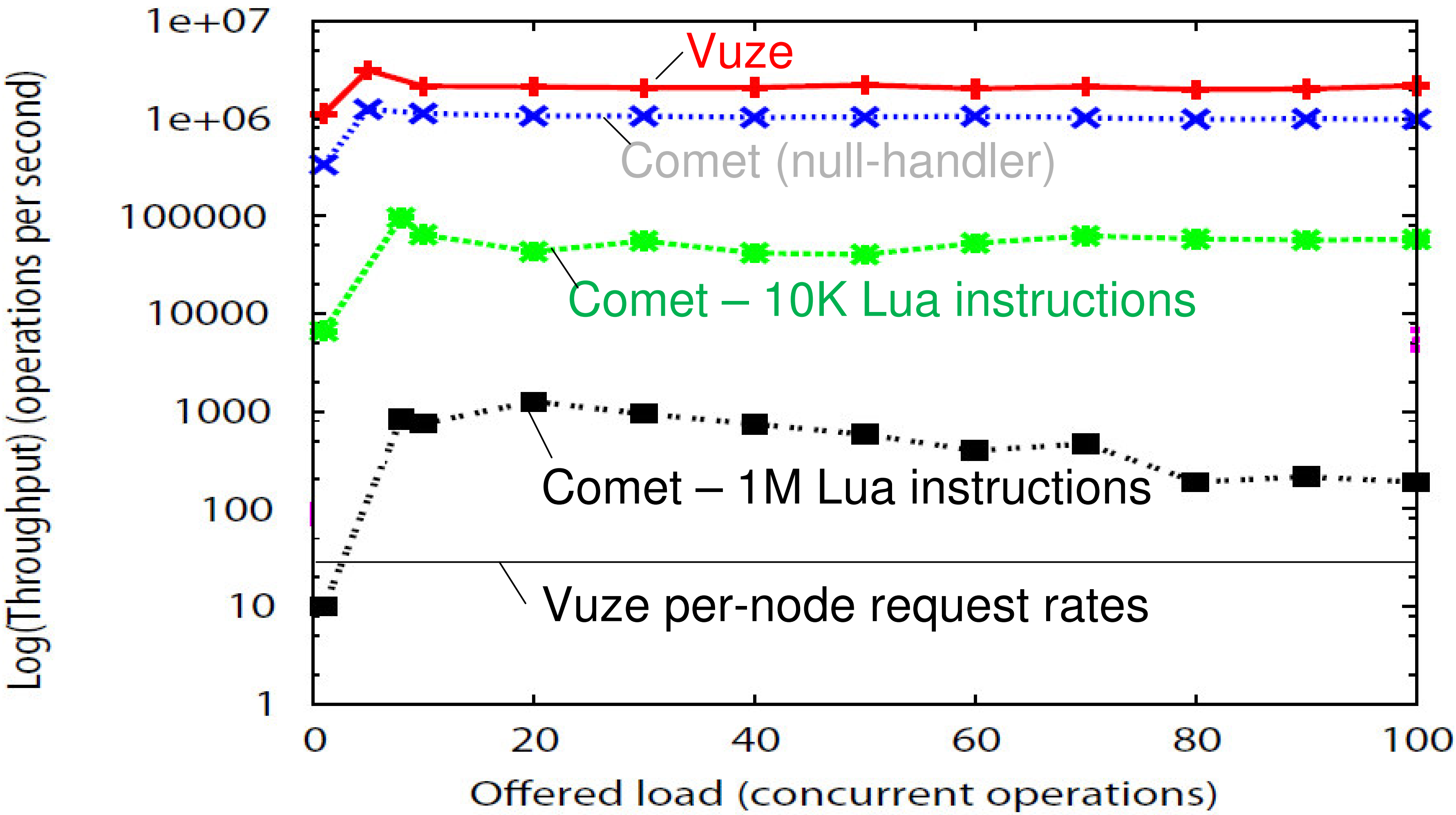
# Comet Throughput



# Memory Footprint



# Comet Throughput



# Related Work

- Extensible systems:
  - Active networks, active messages, extensible OSes (e.g., SPIN), database triggers, extensible routers (e.g., Click), extensible Web crawlers (e.g., xCrawler)
  - Comet has similar extensibility goals
  - But the application domain is different: we build extensible key/value stores
- Object-oriented databases (e.g., Thor):
  - Application domain, environment, and trust are different
- Bigtable Coprocessors:
  - Similar in the idea of pushing code into the storage system
  - Different in environment and trust

# Related Work

- The Ephemerizer (next slide)
- Forward-secure encryption
  - Protects against retroactive data disclosures if attacker obtains the *current* version of the user's keys, but not if he gets keys from *before*
  - Vanish protects even if attacker gets user's keys from before (e.g., from full-disk backups systems via subpoenas)
- Key-insulated and intrusion-resilient systems
  - Same as above + trusted agents or hardware
- Exposure-resilient crypto
  - Assumes that attacker can only see parts of the key
- Self-destructing email services
  - Trust issue: users may be reluctant to trust centralized services
  - In general, only support one type of data (emails)

# Vanish Vs. The Ephemerizer

Similarities:

- Same end-goal: make data self-destruct

Differences:

- Trust models:
  - Vanish shuns trust in any centralized systems
  - The Ephemerizer requires user to trust centralized services that take care of key management for him
- Deployability models:
  - Vanish is readily deployable, as it “parasitically” piggybacks on existing distributed systems
  - The Ephemerizer requires deployment of a dedicated service
- Evaluation and implementation levels:
  - We built and evaluated Vanish

# The ASO Sandbox

1. Limit ASO's knowledge and access
   - We use a language-based sandbox
     - Based on Lua
       - A small, fast, scripting language for coding extensions
       - Used for SimCity, Photoshop, World of Warcraft, ….
     - We made the sandbox as small as possible (<5,000 LOC)
       - We removed unneeded functions from Lua

2. Limit ASO's resource consumption
   - Limit per-handler bytecode instructions and memory
   - Rate-limit incoming and outgoing ASO requests

3. Restrict ASO's DHT interaction
   - Prevent traffic amplification and DDoS attacks
   - ASOs can talk only to their neighbors, no recursive requests

# Closest Related Work

Active Networks:

- Similar motivation and goals
  - We need extensibility; it's hard to deploy changes to infrastructures that we don't control
- Different application domains, hence different design
  - Networks vs. storage systems
  - The API, extensibility points, and sandboxing are different

DB Triggers and Bigtable Coprocessors:

- Similar extensibility goals
- Different environments and trust models, hence different design

# Evaluation Highlights

- Small handler code
  - 100s – 10K Lua bytecode instructions
- Small memory overhead
  - Per-ASO memory consumption: 1KB – 100KB
  - 27% overhead for maximum per-node load in Vuze today
- Small latency overhead
  - Handler Comet delays: microseconds – milliseconds
  - Irrelevant compared to Vuze's lookup latencies (seconds)
- Irrelevant throughout overhead
  - But Comet can handle over three orders of magnitude more requests than the current Vuze request rates

# Comet Flexibility and Limitations

- Flexibility / security / lightweightness tradeoff:
  - Our current design favors security and lightweightness
  - Our design supports a variety of relatively powerful applications
  - Still, more experience is needed to find the “right” tradeoff
- Example limitations:
  - Internet network delay measurements (requires network)
  - Persistent objects (require file I/O)
  - Debugging DHT performance bottlenecks (requires CPU info)
- Signed ASOs can address limitations

# Alternative Designs

- Smarter end applications (end-to-end argument)
  - Sometimes works, but with efforts
  - Other times, simply impossible
- Implement all of the required features in the DHT and expose a richer API
  - Always possible, but one needs to predict all possible needs
  - Debugging and experimentation are key Comet advantages
- Associate the code with keys instead of data
  - Advantage over Comet: continue to trigger
  - Disadvantage over Comet: multi-trigger semantics is unclear
- Overall, we believe that Comet is well suited for DHTs

# “Active” S3: What might be different?

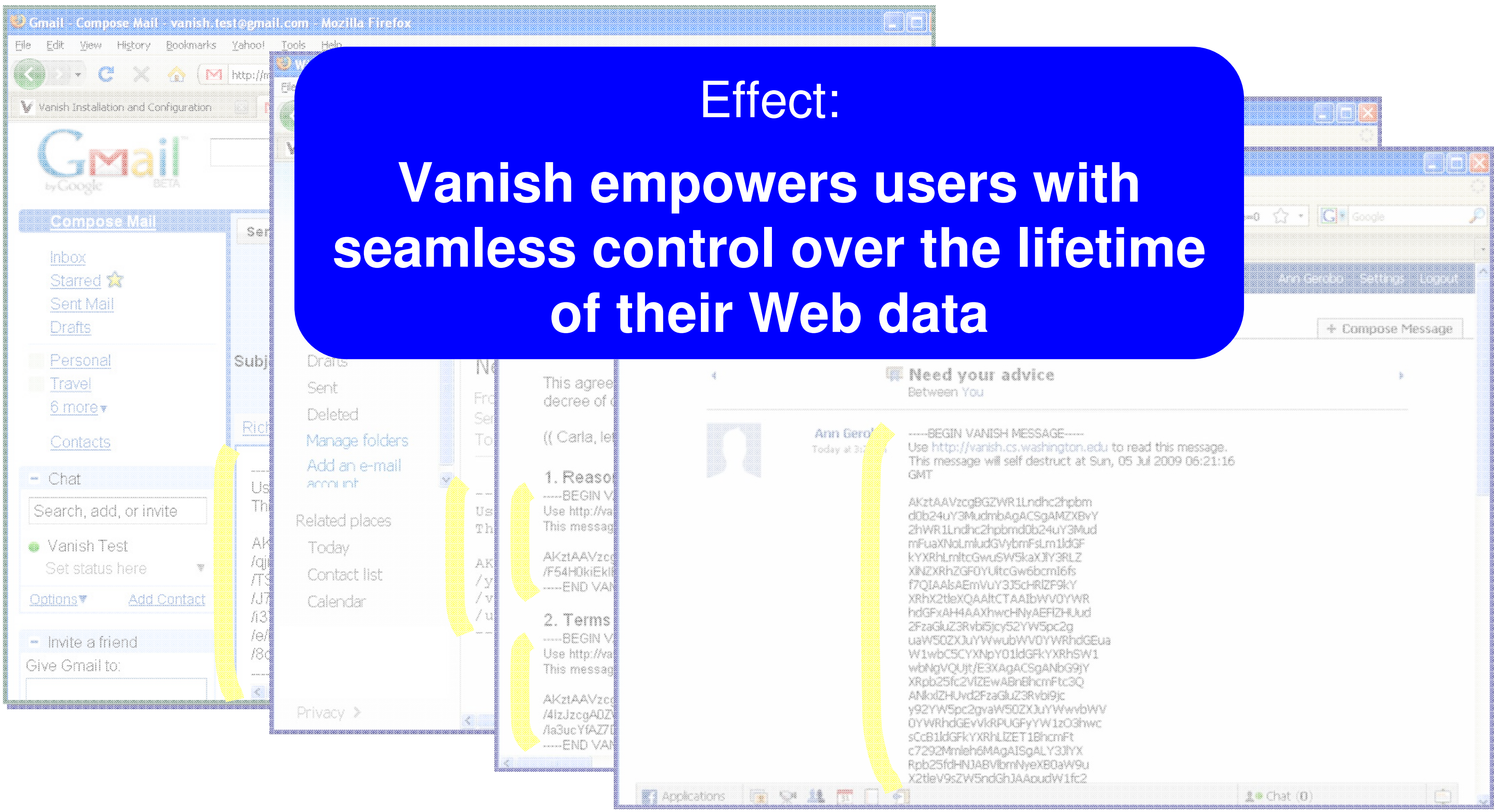
- At what level do we add extensibility?
  - S3 abstractions are quite different from DHT abstractions
  - Buckets, hierarchical index space, user accounts
- What’s the right flexibility/security tradeoff there?
  - Must take into account the datacenter applications, which are very different from DHT applications
- What are the right sandboxing mechanisms?
- Possible first step:
  - Look at Google CoProcessors and sandbox them

# Vuze and Comet Workloads

- Vuze per-node workloads:
  - Request rate: 30 – 100 requests/s
  - Number of values: 400 – 30,000 values
- Comet per-handler workloads:
  - Lua instructions: 100 – 10K
  - Memory footprint: 1K – 100K
- Comet **`onTimer`** interval: 20 min

# Firefox Plugin For Vanishing Web Data

- Encapsulate text in any text area in self-destructing data




Effect:

**Vanish empowers users with seamless control over the lifetime of their Web data**

# How it works

- Over the last year we have designed several possible solutions.
- All solutions use highly distributed storage systems with multiple trust domains, including:
  - Distributed Hash Tables (DHTs)
  - Collections of globally distributed services
  - A hybrid approach with multiple types of storage systems, each with different security and trust properties

I'll (try to ☺) describe one solution….using DHTs.

# Retroactive Attack

- Discloses old copies of sensitive data months or years after data creation (and attempted deletion)
- Retroactive attacks have become commonplace:
  - Hackers
  - Subpoenas
  - Misconfigurations
  - Laptops seized at borders
  - Device theft
  - Carelessness
  - …

# Roxana Geambasu

http://www.cs.washington.edu/homes/roxana/