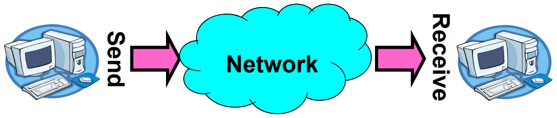
CS162
Operating Systems and Systems Programming
Lecture 26

Trusted Execution, Distributed File Systems
Global Data Plane

April 30th, 2024
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Recall: Distributed Applications Build With Messages

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set

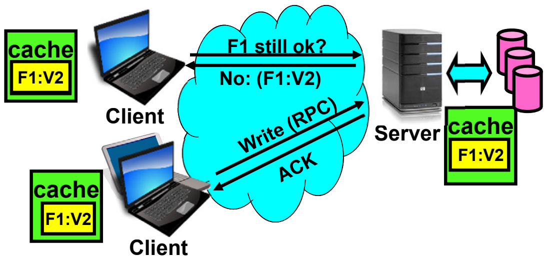


  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (mbox): temporary holding area for messages
    - » Includes both destination location and queue
  - Send(message,mbox)
    - » Send message to remote mailbox identified by mbox
  - Receive(buffer,mbox)
    - » Wait until mbox has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them

---

Recall: NFS Cache consistency

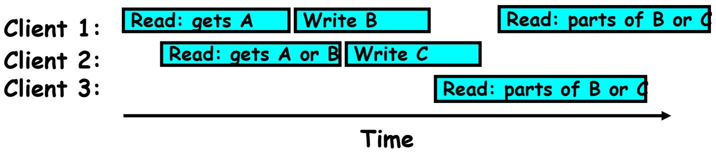
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout is tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"



- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - As a result, do not get partial writes: all or nothing!
    - Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

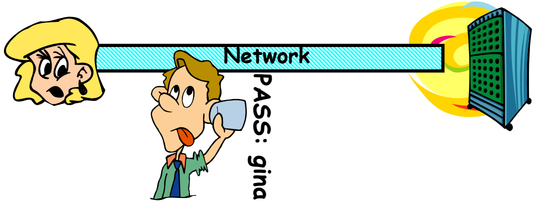
Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - Get file from server, set up callback with server
  - On write followed by close:
    - Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- AFS Pro: Relative to NFS, less server load:
  - Disk as cache ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation

Quick Security Primer

Authentication in Distributed Systems

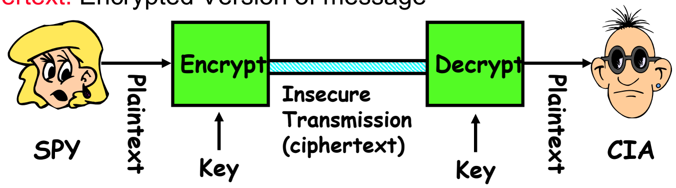
- What if identity must be established across network?



  - Need way to prevent exposure of information while still proving identity to remote system
  - Many of the original UNIX tools sent passwords over the wire "in clear text"
    - E.g.: telnet, ftp, yp (yellow pages, for distributed login)
    - Result: Snooping programs widespread
- What do we need? Cannot rely on physical security!
  - Encryption: Privacy, restrict receivers
  - Authentication: Remote Authenticity, restrict senders

Private Key Cryptography

- Private Key (Symmetric) Encryption:
  - Single key used for both encryption and decryption
- Plaintext: Unencrypted Version of message
- Ciphertext: Encrypted Version of message

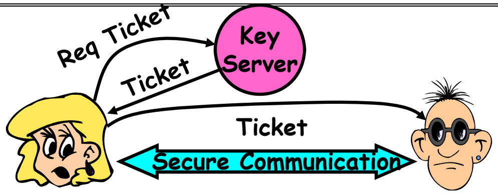


- Important properties
  - Can't derive plain text from ciphertext (decode) without access to key
  - Can't derive key from plain text and ciphertext
  - As long as password stays secret, get both secrecy and authentication
- Symmetric Key Algorithms: DES, Triple-DES, AES

Key Distribution

- How do you get shared secret to both places?
  - For instance: how do you send authenticated, secret mail to someone who you have never met?
  - Must negotiate key over private channel
    - » Exchange code book
    - » Key cards/memory stick/others
- Third Party: Authentication Server (like Kerberos)
  - Notation:
    - » K_{xy} is key for talking between x and y
    - » $(\ldots)^K$ means encrypt message (…) with the key K
    - » Clients: A and B, Authentication server S
  - A asks server for key:
    - » A→S: [Hi! I'd like a key for talking between A and B]
    - » Not encrypted. Others can find out if A and B are talking
  - Server returns *session* key encrypted using B's key
    - » S→A: Message [Use K_{ab} (This is A! Use K_{ab})^{Ksb}] ^{Ksa}
    - » This allows A to know, "S said use this key"
  - Whenever A wants to talk with B
    - » A→B: Ticket [This is A! Use K_{ab}]^{Ksb}
    - » Now, B knows that K_{ab} is sanctioned by S

Authentication Server Continued [Kerberos]



- Details
  - Both A and B use passwords (shared with key server) to decrypt return from key servers
  - Add in timestamps to limit how long tickets will be used to prevent attacker from replaying messages later
  - Also have to include encrypted checksums (hashed version of message) to prevent malicious user from inserting things into messages/changing messages
  - Want to minimize # times A types in password
    - » A→S (Give me temporary secret)
    - » S→A (Use $K_{temp\text{-}sa}$ for next 8 hours)^{Ksa}
    - » Can now use $K_{temp\text{-}sa}$ in place of K_{sa} in prototcol

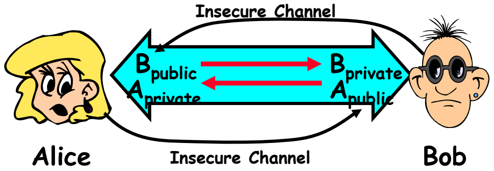
---

Public Key Encryption

- Can we perform key distribution without an authentication server?
  - Yes. Use a Public-Key Cryptosystem.
- Public Key Details
  - Don't have one key, have two: K_{public}, $K_{private}$
    - » Two keys are mathematically related to one another
    - » Really hard to derive K_{public} from $K_{private}$ and vice versa
  - Forward encryption:
    - » Encrypt: (cleartext)^{Kpublic}= ciphertext$_1$
    - » Decrypt: (ciphertext$_1$)^{Kprivate} = cleartext
  - Reverse encryption:
    - » Encrypt: (cleartext)^{Kprivate} = ciphertext$_2$
    - » Decrypt: (ciphertext$_2$)^{Kpublic} = cleartext
  - Note that ciphertext$_1$ ≠ ciphertext$_2$
    - » Can't derive one from the other!
- Public Key Examples:
  - RSA: Rivest, Shamir, and Adleman
    - » K_{public} of form (k_{public}, N), $K_{private}$ of form ($k_{private}$, N)
    - » N = pq. Can break code if know p and q
  - ECC: Elliptic Curve Cryptography
    - » Lower overhead than RSA

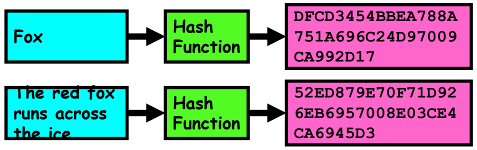
Public Key Encryption Details

- Idea: K_{public} can be made public, keep $K_{private}$ private



- Gives message privacy (restricted receiver):
  - Public keys (secure destination points) can be acquired by anyone/used by anyone
  - Only person with private key can decrypt message
- What about authentication?
  - Use combination of private and public key
  - Alice→Bob: [(I’m Alice)$^{\text{Aprivate}}$ Rest of message]$^{\text{Bpublic}}$
  - Provides restricted sender and receiver
- But: how does Alice know that it was Bob who sent her B_{public}? And vice versa…

Secure Hash Function

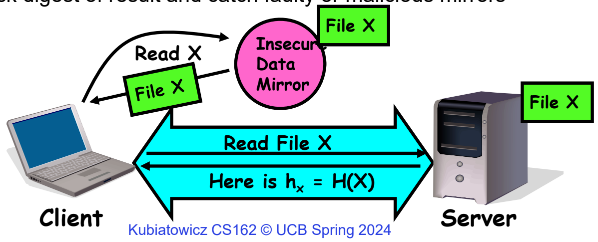


- Hash Function: Short summary of data (message)
  - For instance, $h_1=H(M_1)$ is the hash of message M_1
    - » h_1 fixed length, despite size of message M_1.
    - » Often, h_1 is called the “digest” of M_1.
- Hash function H is considered secure if
  - It is infeasible to find M_2 with $h_1=H(M_2)$; ie. can’t easily find other message with same digest as given message.
  - It is infeasible to locate two messages, m_1 and m_2, which “collide”, i.e. for which $H(m_1) = H(m_2)$
  - A small change in a message changes many bits of digest/can’t tell anything about message given its hash

---

Use of Hash Functions

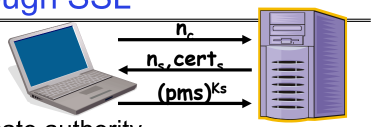
- Several Standard Hash Functions:
  - MD5: 128-bit output
  - SHA-1: 160-bit output, SHA-256: 256-bit output
- Can we use hashing to securely reduce load on server?
  - Yes. Use a series of insecure mirror servers (caches)
  - First, ask server for digest of desired file
    - » Use secure channel with server
  - Then ask mirror server for file
    - » Can be insecure channel
    - » Check digest of result and catch faulty or malicious mirrors



Signatures/Certificate Authorities

- Can use X_{public} for person X to define their identity
  - Presumably they are the only ones who know $X_{private}$.
  - Often, we think of X_{public} as a “principle” (user)
- Suppose we want X to sign message M?
  - Use private key to encrypt the digest, i.e. H(M)$^{\text{Xprivate}}$
  - Send both M and its signature:
    - » Signed message = [M,H(M)$^{\text{Xprivate}}$]
  - Now, anyone can verify that M was signed by X
    - » Simply decrypt the digest with X_{public}
    - » Verify that result matches H(M)
- Now: How do we know that the version of X_{public} that we have is really from X???
  - Answer: Certificate Authority
    - » Examples: Verisign, Entrust, Etc.
  - X goes to organization, presents identifying papers
    - » Organization signs X’s key: [X_{public}, H(X_{public})$^{\text{CAprivate}}$]
    - » Called a “Certificate”
  - Before we use X_{public}, ask X for certificate verifying key
    - » Check that signature over X_{public} produced by trusted authority
- How do we get keys of certificate authority?
  - Compiled into your browser, for instance!

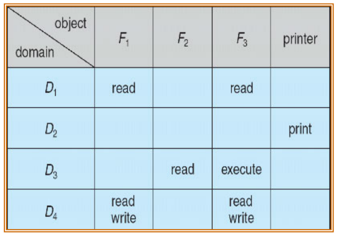
Security through SSL



- SSL Web Protocol
  - Port 443: secure http
  - Use public-key encryption for key-distribution
- Server has a certificate signed by certificate authority
  - Contains server info (organization, IP address, etc)
  - Also contains server's public key and expiration date
- Establishment of Shared, 48-byte "master secret"
  - Client sends 28-byte random value n_c to server
  - Server returns its own 28-byte random value n_s, plus its certificate $cert_s$
  - Client verifies certificate by checking with public key of certificate authority compiled into browser
    - » Also check expiration date
  - Client picks 46-byte "premaster" secret (pms), encrypts it with public key of server, and sends to server
  - Now, both server and client have n_c, n_s, and pms
    - » Each can compute 48-byte master secret using one-way and collision-resistant function on three values
    - » Random "nonces" n_c and n_s make sure master secret fresh

---

Authorization: Who Can Do What?



object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- How do we decide who is authorized to do actions in the system?
- Access Control Matrix: all permissions in the system
  - Resources across top
    - » Files, Devices, etc…
  - Domains in columns
    - » A domain might be a user or a group of permissions
    - » E.g. above: User D_3 can read F_2 or execute F_3
  - In practice, table would be huge and sparse!
- Two approaches to implementation
  - Access Control Lists: store permissions with each object
    - » Still might be lots of users!
    - » UNIX limits each file to: r,w,x for owner, group, world
    - » More recent systems allow definition of groups of users and permissions for each group
  - Capability List: each process tracks objects has permission to touch
    - » Popular in the past, idea out of favor today
    - » Consider page table: Each process has list of pages it has access to, not each page has list of processes …

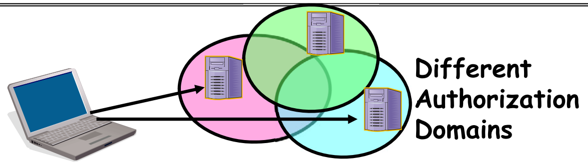
How fine-grained should access control be?

- Example of the problem:
  - Suppose you buy a copy of a new game from "Joe's Game World" and then run it.
  - It's running with your userid
    - » It removes all the files you own, including the project due the next day…
- How can you prevent this?
  - Have to run the program under some userid.
    - » Could create a second games userid for the user, which has no write privileges.
    - » Like the "nobody" userid in UNIX – can't do much
  - But what if the game needs to write out a file recording scores?
    - » Would need to give write privileges to one particular file (or directory) to your games userid.
  - But what about non-game programs you want to use, such as Quicken?
    - » Now you need to create your own private quicken userid, if you want to make sure tha the copy of Quicken you bought can't corrupt non-quicken-related files
  - But – how to get this right??? Pretty complex…

Authorization Continued

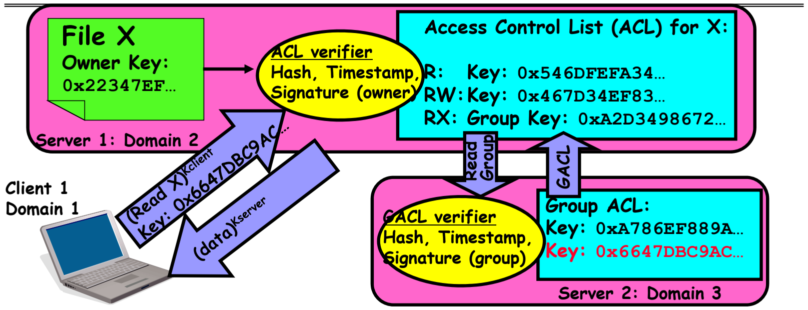
- Principle of least privilege: programs, users, and systems should get only enough privileges to perform their tasks
  - Very hard to do in practice
    - » How do you figure out what the minimum set of privileges is needed to run your programs?
  - People often run at higher privilege then necessary
    - » Such as the "administrator" privilege under windows
- One solution: Signed Software
  - Only use software from sources that you trust, thereby dealing with the problem by means of authentication
  - Fine for big, established firms such as Microsoft, since they can make their signing keys well known and people trust them
    - » Actually, not always fine: recently, one of Microsoft's signing keys was compromised, leading to malicious software that looked valid
  - What about new startups?
    - » Who "validates" them?
    - » How easy is it to fool them?

How to perform Authorization for Distributed Systems?



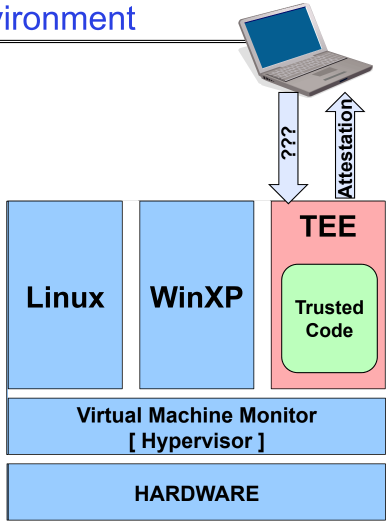
- Issues: Are all user names in world unique?
  - No! They only have small number of characters
    - kubi@mit.edu → kubitron@lcs.mit.edu → kubitron@cs.berkeley.edu
    - However, someone thought their friend was kubi@mit.edu and I got very private email intended for someone else…
  - Need something better, more unique to identify person
- Suppose want to connect with any server at any time?
  - Need an account on every machine! (possibly with different user name for each account)
  - OR: Need to use something more universal as identity
    - Public Keys! (Called "Principles")
    - People *are* their public keys

Distributed Access Control



- Distributed Access Control List (ACL)
  - Contains list of attributes (Read, Write, Execute, etc) with attached identities (Here, we show public keys)
    - ACLs signed by owner of file, only changeable by owner
    - Group lists signed by group key
  - ACLs can be on different servers than data
    - Signatures allow us to validate them
    - ACLs could even be stored separately from verifiers

Trusted Execution Environment



- Simple Hardware with single OS
  - What we have been talking about all term!
- Virtual machines
  - Multiplex different OSes on single machine
  - Many techniques, including dynamic compilation and direct hardware support (domain "-1")
  - Need way to fool OS code into thinking it has complete control of machine!
- What if you don't trust the OS or hypervisor not to leak your information?
  - Worried about compromised OS
  - Don't trust service provider (i.e. Google, Amazon)
- Trusted Execution Environment (TEE)
  - Hardware support to prevent OS or external actors from observing execution
  - Client can get hardware proof that trusted code is actual code we expect! [Attestation]

Chord and Distributed Storage

What about: Sharing Data, rather than Files ?

- Key:Value stores are used everywhere
- Native in many programming languages
  - Associative Arrays in Perl
  - Dictionaries in Python
  - Maps in Go
  - …
- What about a collaborative key-value store rather than message passing or file sharing?
- Can we make it scalable and reliable?

Key Value Storage

Simple interface

- **`put(key, value);`** `// Insert/write "value" associated with key`
- **`get(key);`** `// Retrieve/read value associated with key`

Why Key Value Storage?

- Easy to Scale
  - Handle huge volumes of data (e.g., petabytes)
  - Uniform items: distribute easily and roughly equally across many machines
- Simple consistency properties
- Used as a simpler but more scalable "database"
  - Or as a building block for a more capable DB

Key Values: Examples

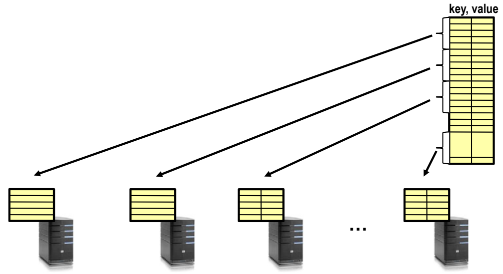
- Amazon:
  - Key: customerID
  - Value: customer profile (e.g redit card, ..)
- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)
- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

Key-value storage systems in real life

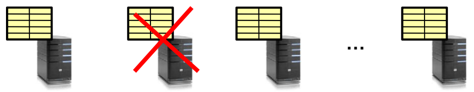
- **Amazon**
  - – DynamoDB: internal key value store used to power Amazon.com (shopping cart)
  - – Simple Storage System (S3)
- **BigTable/HBase/Hypertable:** distributed, scalable data storage
- **Cassandra**: “distributed data management system” (developed by Facebook)
- **Memcached:** in-memory key-value store for small chunks of arbitrary data (strings, objects)
- **eDonkey/eMule:** peer-to-peer sharing system
- …

Key Value Store

- Also called Distributed Hash Tables (DHT)
- Main idea: simplify storage interface (i.e. put/get), then partition set of key-values across many machines



Challenges



- **Scalability:**
  - – Need to scale to thousands of machines
  - – Need to allow easy addition of new machines
- **Fault Tolerance:** handle machine failures without losing data and without degradation in performance
- **Consistency:** maintain data consistency in face of node failures and message losses
- **Heterogeneity** (if deployed as peer-to-peer systems):
  - – Latency: 1ms to 1000ms
  - – Bandwidth: 32Kb/s to 100Mb/s

Important Questions

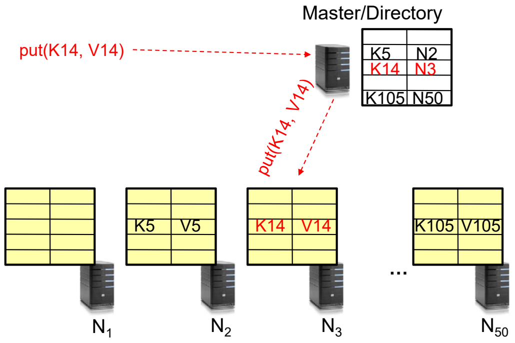
- **put(key, value)**:
  - – where do you store a new (key, value) tuple?
- **get(key)**:
  - – where is the value associated with a given “key” stored?
- And, do the above while providing
  - – Scalability
  - – Fault Tolerance
  - – Consistency

How to solve the “where?”

- Hashing to map key space ⇒ location
  - But what if you don’t know all the nodes that are participating?
  - Perhaps they come and go …
  - What if some keys are really popular?
- Lookup
  - Hmm, won’t this be a bottleneck and single point of failure?

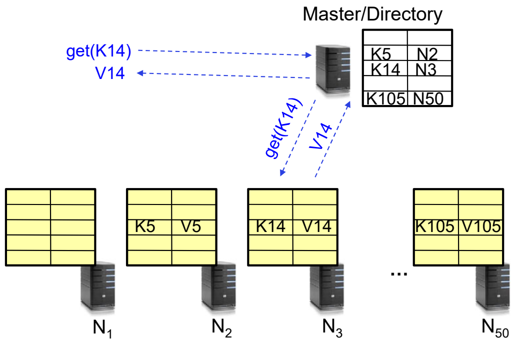
Recursive Directory Architecture (put)

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



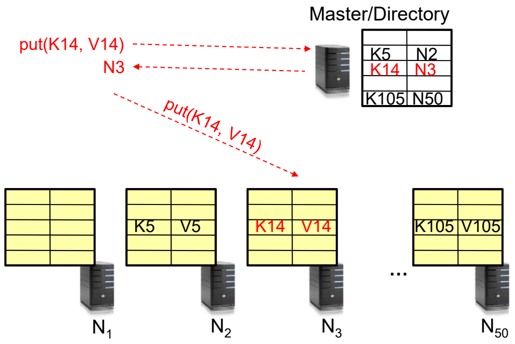
Recursive Directory Architecture (get)

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**



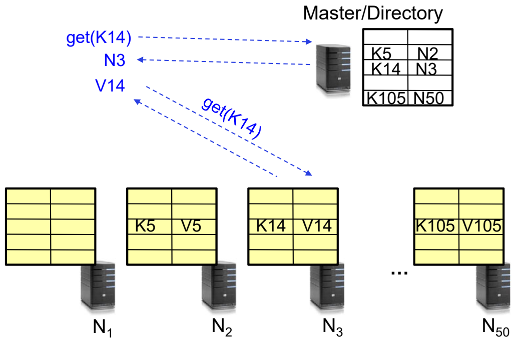
Iterative Directory Architecture (put)

- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  - Return node to requester and let requester contact node

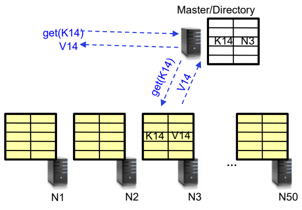


Iterative Directory Architecture (get)

- Having the master relay the requests → **recursive query**
- Another method: **iterative query** (this slide)
  - – Return node to requester and let requester contact node

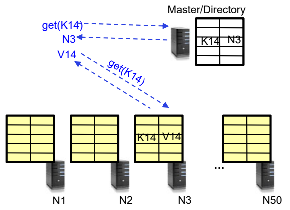


Iterative vs. Recursive Query



Recursive

- + Faster, as directory server is typically close to storage nodes
- + Easier for consistency: directory can enforce an order for all puts and gets
- - Directory is a performance bottleneck

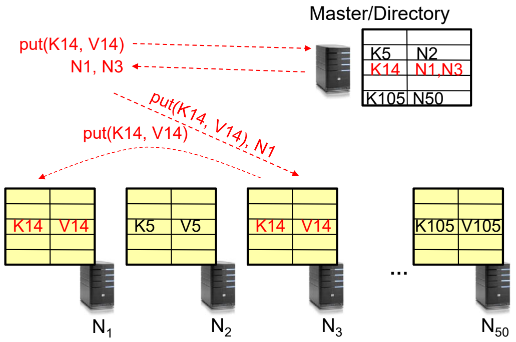


Iterative

- + More scalable, clients do more work
- - Harder to enforce consistency

Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures



Scalability

- Storage: use more nodes
- Number of requests:
  - – Can serve requests from all nodes on which a value is stored in parallel
  - – Master can replicate a popular value on more nodes
- Master/directory scalability:
  - – Replicate it
  - – Partition it, so different keys are served by different masters/directories
    - » How do you partition?

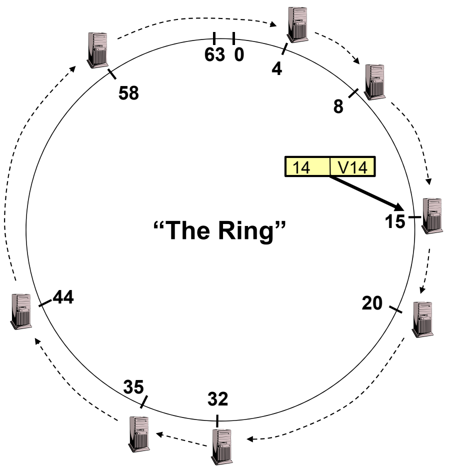
Scaling Up Directory

- Challenge:
  - Directory contains a number of entries equal to number of (key, value) tuples in the system
  - Can be tens or hundreds of billions of entries in the system!
- Solution: **Consistent Hashing**
  - **Provides mechanism to divide [key,value] pairs amongst a (potentially large!) set of machines (nodes) on network**
- Associate to each node a unique *id* in an *uni*-dimensional space $0..2^m-1$
  ⇒ Wraps around: Call this "the ring!"
  - Partition this space across *n* machines
  - Assume keys are in same uni-dimensional space
  - Each [Key, Value] is stored at the node with the smallest ID larger than Key

---

Key to Node Mapping Example

- Paritioning example with m = 6 → ID space: 0..63
  - Node 8 maps keys [5,8]
  - Node 15 maps keys [9,15]
  - Node 20 maps keys [16, 20]
  - …
  - Node 4 maps keys [59, 4]
- For this example, the mapping [14, V14] maps to node with ID=15
  - Node with smallest ID larger than 14 (the key)
- In practice, m=256 or more!
  - Uses cryptographically secure hash such as SHA-256 to generate the node IDs

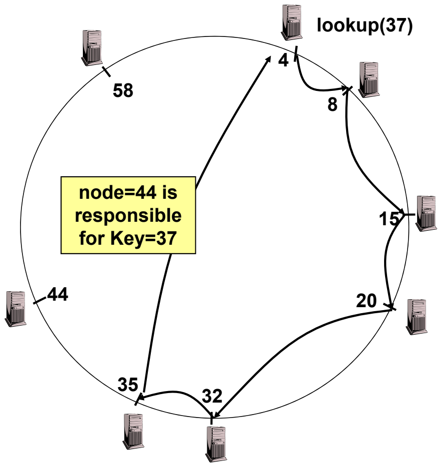


Chord: Distributed Lookup (Directory) Service

- "Chord" is a Distributed Lookup Service
  - Designed at MIT and here at Berkeley (Ion Stoica among others)
  - Simplest and cleanest algorithm for distributed storage
    - » Serves as comparison point for other optims
- Import aspect of the design space:
  - Decouple correctness from efficiency
  - Combined *Directory* and *Storage*
- Properties
  - Correctness:
    - » Each node needs to know about neighbors on ring (one predecessor and one successor)
    - » Connected rings will perform their task correctly
  - Performance:
    - » Each node needs to know about O(log(*M*)), where *M* is the total number of nodes
    - » Guarantees that a tuple is found in O(log(*M*)) steps
- Many other *Structured, Peer-to-Peer* lookup services:
  - CAN, Tapestry, Pastry, Bamboo, Kademlia, …
  - Several designed here at Berkeley!

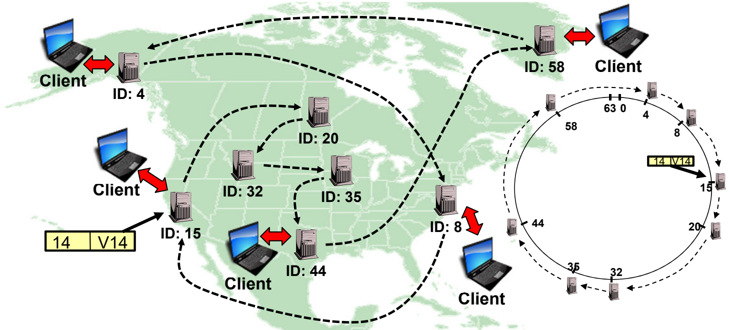
Chord's Lookup Mechanism: Routing!

- Each node maintains pointer to its successor
- Route packet (Key, Value) to the node responsible for ID using successor pointers
  - E.g., node=4 lookups for node responsible for Key=37
- Worst-case (correct) lookup is O(n)
  - But much better normal lookup time is O(log n)
  - Dynamic performance optimization (finger table mechanism)
    - » More later!!!



node=44 is responsible for Key=37

But what does this really mean??



- Node names intentionally scrambled WRT geography!
  - Node IDs generated by secure hashes over metadata
    - Including things like the IP address
  - This geographic scrambling spreads load and avoids hotspots
- Clients access distributed storage through any member of the network

---

Stabilization Procedure

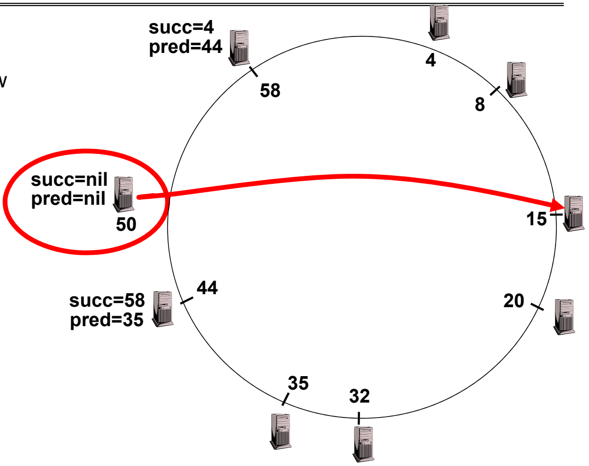
- Periodic operation performed by each node n to maintain its successor when new nodes join the system
  - The primary Correctness constraint

```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
      succ = x;      // if x better successor, update
  succ.notify(n); // n tells successor about itself

n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
      pred = n';      // if n' is better predecessor, update
```

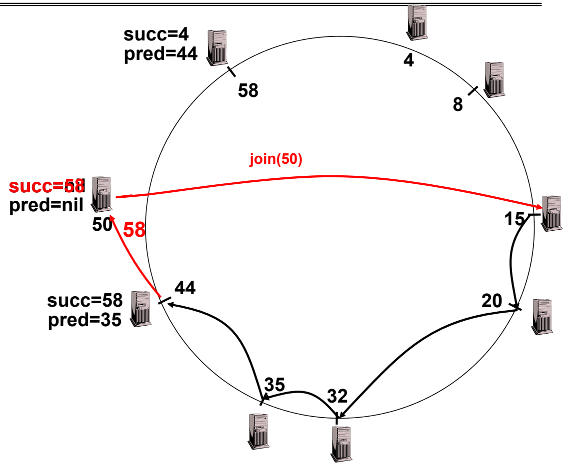
Joining Operation

- Node with id=50 joins the ring
- Node 50 must know at least one node already in system
  - Assume known node is 15



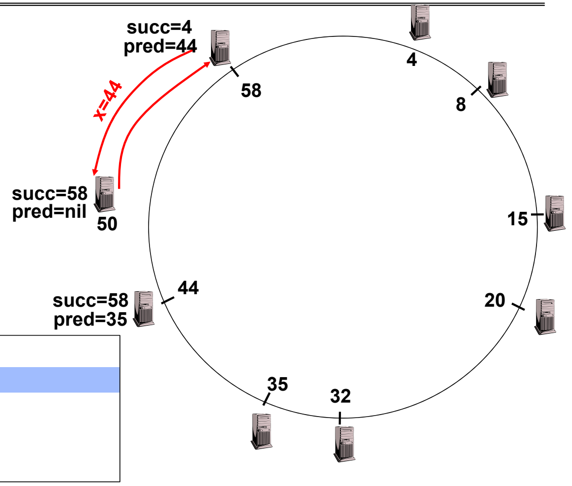
Joining Operation

- n=50 sends join(50) to node 15
  - Join propagated around ring!
- n=44 returns node 58
- n=50 updates its successor to 58



Joining Operation

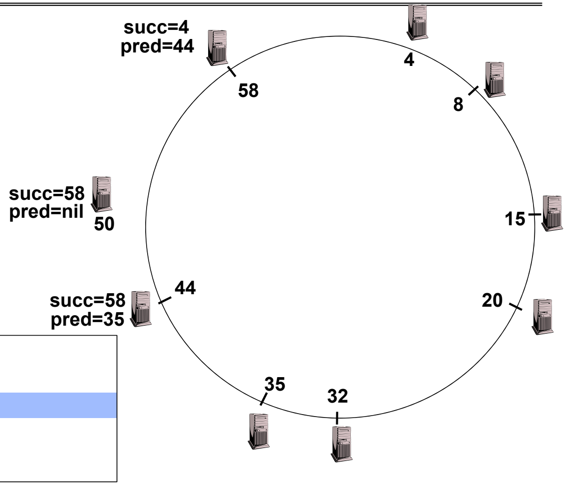
- n=50 executes stabilize()
- n’s successor (58) returns x = 44



```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

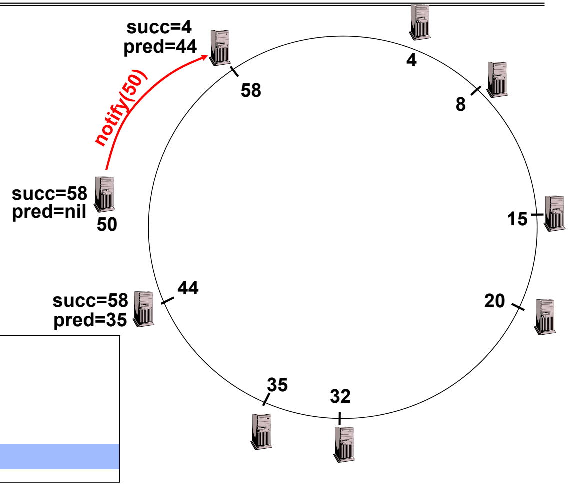
- n=50 executes stabilize()
  - x = 44
  - succ = 58



```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

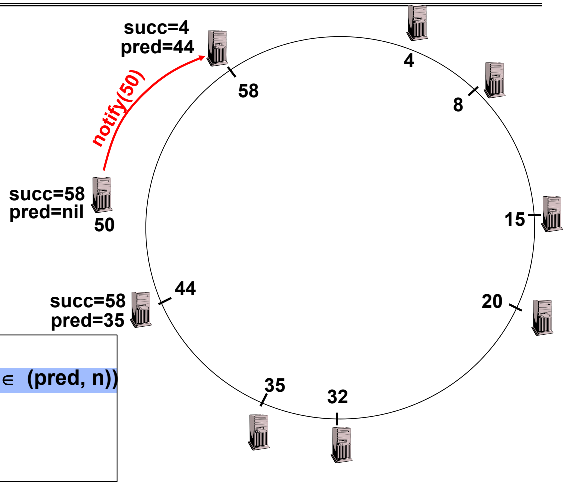
- n=50 executes stabilize()
  - x = 44
  - succ = 58
- n=50 sends to it’s successor (58) notify(50)



```
n.stabilize()
  x = succ.pred;
  if (x ∈(n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

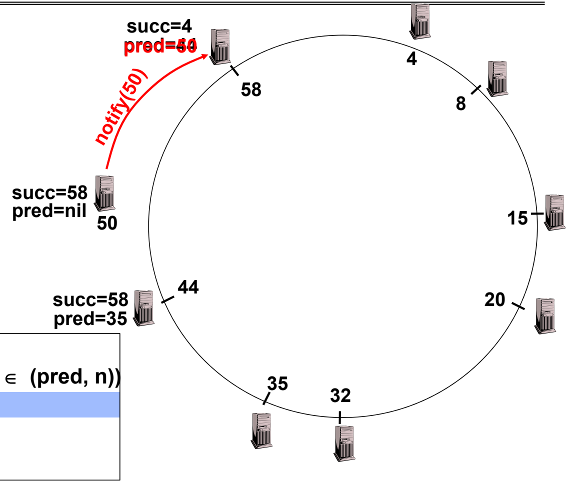
- n=58 executes notify(50)
  - pred = 44
  - n’ = 50



```
n.notify(n’)
  if (pred = nil or n’ ∈ (pred, n))
    pred = n’
```

Joining Operation

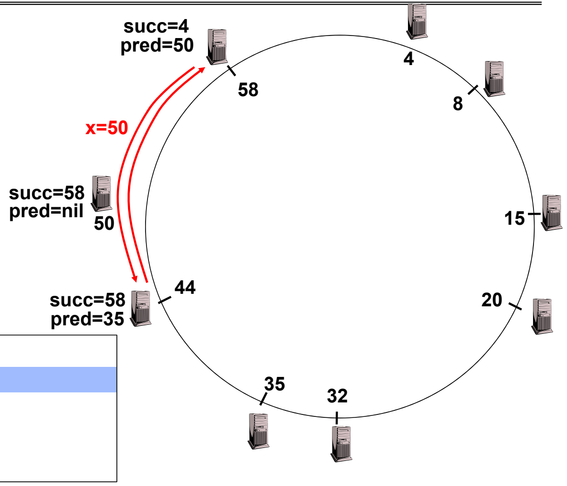
- n=58 executes notify(50)
  - pred = 44
  - n' = 50
- set pred = 50



```
n.notify(n')
  if (pred = nil or n' ∈ (pred, n))
    pred = n'
```

Joining Operation

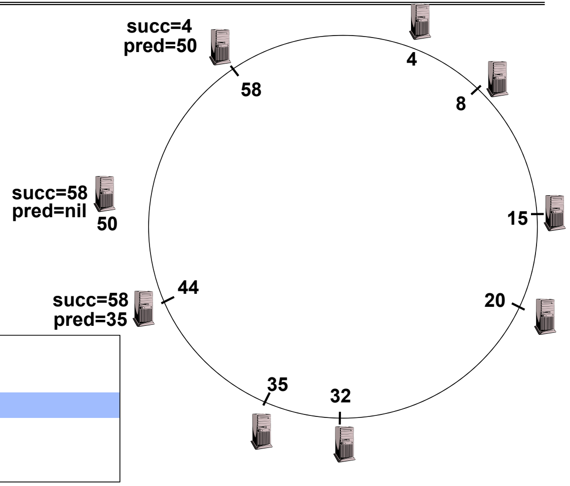
- n=44 executes stabilize()
- n's successor (58) returns x=50



```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

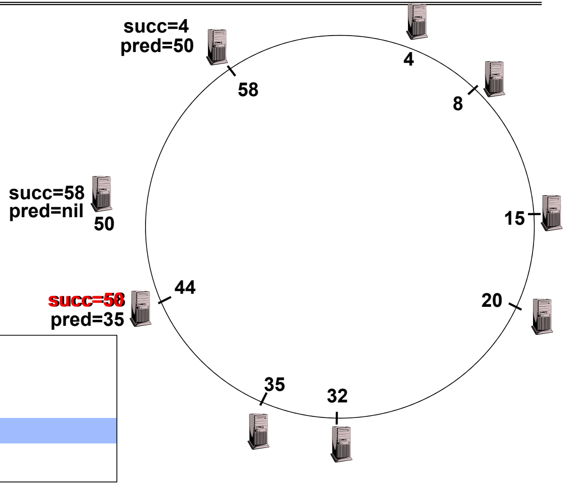
- n=44 executes stabilize()
  - x=50
  - succ=58



```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

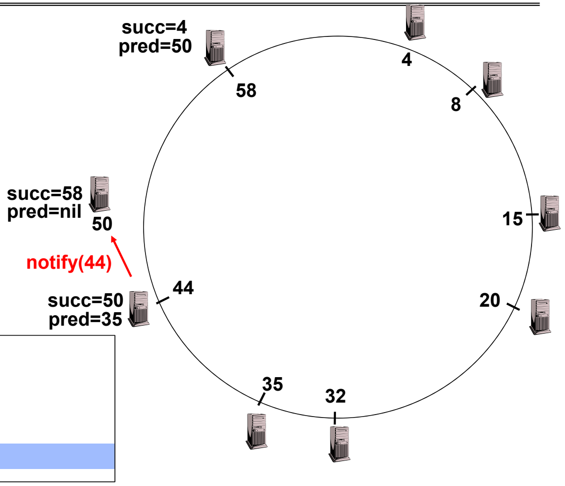
- n=44 executes stabilize()
  - x=50
  - succ=58
- n=44 sets succ=50



```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
    succ = x;
  succ.notify(n);
```

Joining Operation

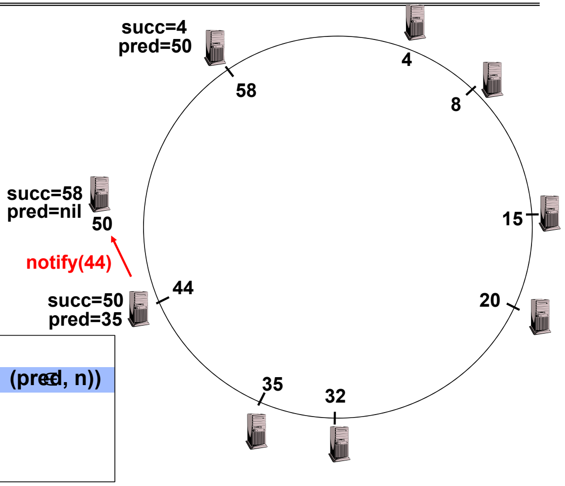
- n=44 executes stabilize()
- n=44 sends notify(44) to its successor



```
n.stabilize()
  x = succ.pred;
  if (x ∈ (n, succ))
     succ = x;
  succ.notify(n);
```

Joining Operation

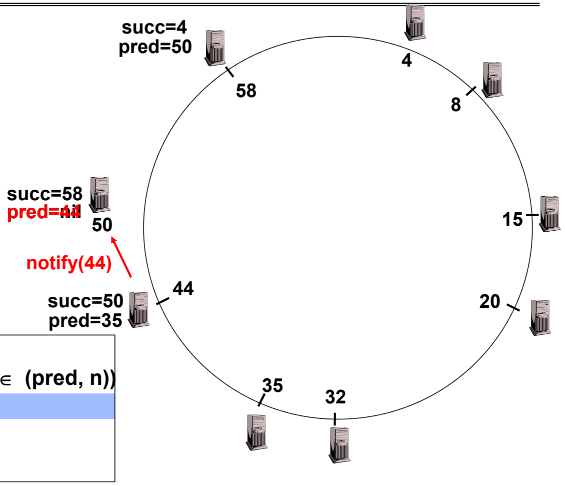
- n=50 executes notify(44)
  - pred=nil



```
n.notify(n’)
  if (pred = nil or n’ ∈ (pred, n))
     pred = n’
```

Joining Operation

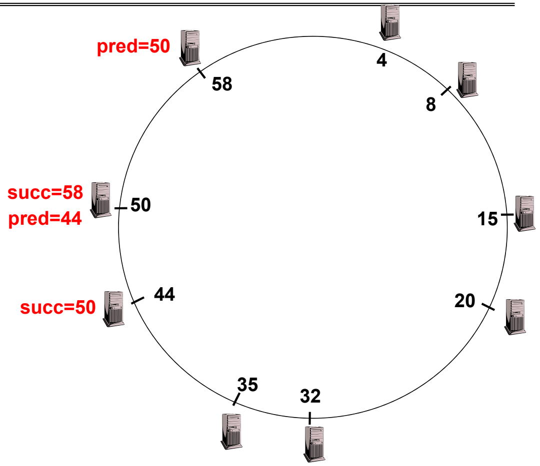
- n=50 executes notify(44)
  - pred=nil
- n=50 sets pred=44



```
n.notify(n’)
  if (pred = nil or n’ ∈ (pred, n))
     pred = n’
```

Joining Operation (cont’d)

- This completes the joining operation!
- The same stabilizing process will deal with failed nodes by reconnecting the ring
- What if 2 or more nodes in a row fail?
  - Keep track of more neighbors!
  - Called the “leaf set”

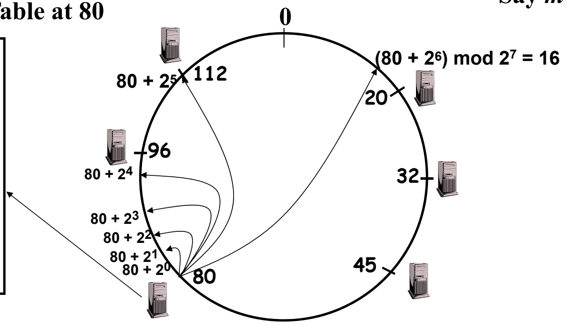


Achieving Efficiency: *finger tables*

Finger Table at 80

i	*ft[i]*
0	96
1	96
2	96
3	96
4	96
5	112
6	20

Say *m=7*



*i*th entry at peer with id *n* is first peer with id >= $n + 2^i \pmod{2^m}$

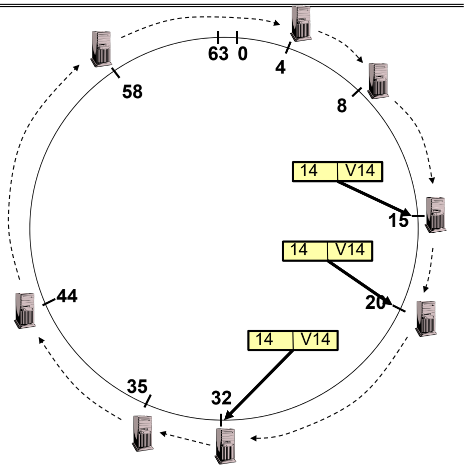
Achieving Fault Tolerance for Lookup Service

- To improve robustness each node maintains the k (> 1) immediate successors instead of only one successor
  - Again – called the "leaf set"
  - In the pred() reply message, node A can send its k-1 successors to its predecessor B
  - Upon receiving pred() message, B can update its successor list by concatenating the successor list received from A with its own list
- If k = log(M), lookup operation works with high probability even if half of nodes fail, where M is number of nodes in the system

---

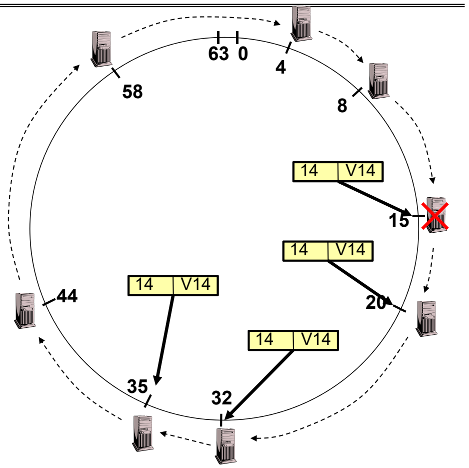
Storage Fault Tolerance

- Replicate tuples on successor nodes
- Example: replicate (K14, V14) on nodes 20 and 32

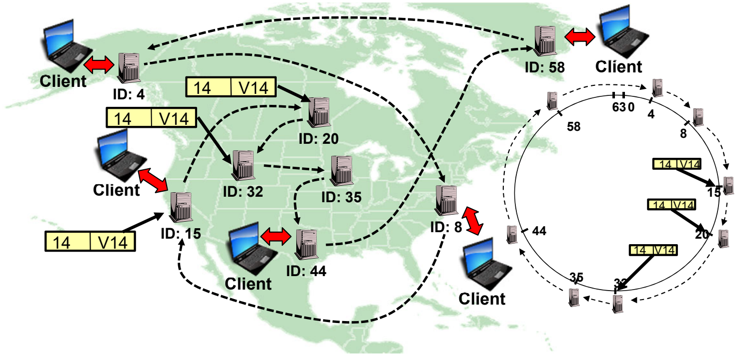


Storage Fault Tolerance

- If node 15 fails, no reconfiguration needed
  - Still have two replicas
  - All lookups will be correctly routed after stabilization
- Will need to add a new replica on node 35



Replication in Physical Space



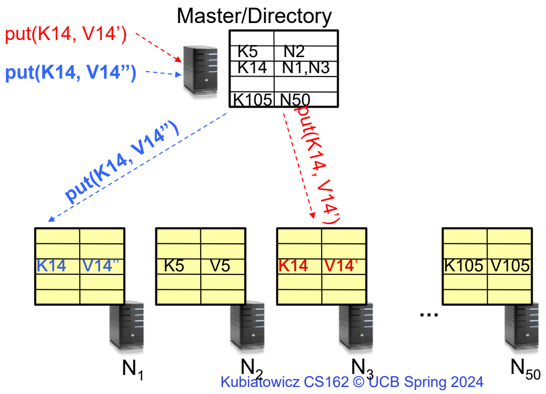
- Replicating in Adjacent nodes of virtual space ⇒ Geographic Separation in physical space
  - Avoids single-points of failure through randomness
  - More nodes, more replication, more geographic spread

Consistency

- Need to make sure that a value is replicated correctly
- How do you know a value has been replicated on every node?
  - Wait for acknowledgements from every node
- What happens if a node fails during replication?
  - Pick another node and try again
- What happens if a node is slow?
  - Slow down the entire put()? Pick another node?
- In general, with multiple replicas
  - Slow puts and fast gets

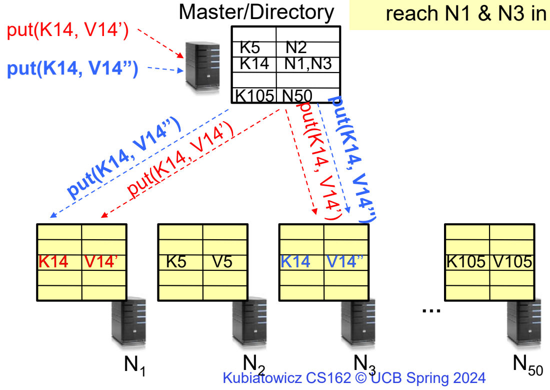
Consistency (cont’d)

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



Consistency (cont’d)

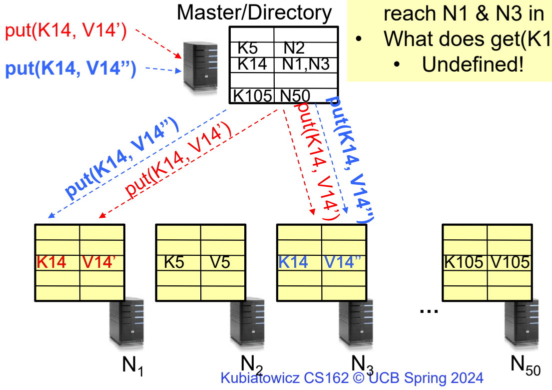
- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14’) and put(K14, V14’’) reach N1 & N3 in reverse order!

Consistency (cont’d)

- If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order



- put(K14, V14’) and put(K14, V14’’) reach N1 & N3 in reverse order!
- What does get(K14) return?
  - Undefined!

Large Variety of Consistency Models

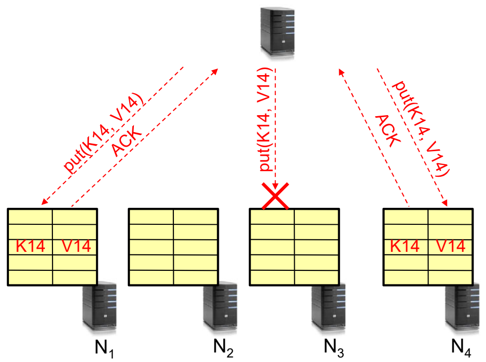
- Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
  - Think “one updated at a time”
  - Transactions
- Eventual consistency: given enough time all updates will propagate through the system
  - One of the weakest form of consistency; used by many systems in practice
  - Must eventually converge on single value/key (coherence)
- And many others: causal consistency, sequential consistency, strong consistency, …

Quorum Consensus

- Improve put() and get() operation performance
  - In the presence of replication!
- Define a replica set of size N
  - put() waits for acknowledgements from at least W replicas
    - » Different updates need to be differentiated by something monotonically increasing like a timestamp
    - » Allows us to replace old values with updated ones
  - get() waits for responses from at least R replicas
  - W+R > N
- Why does it work?
  - There is at least one node that contains the update
- Why might you use W+R > N+1?

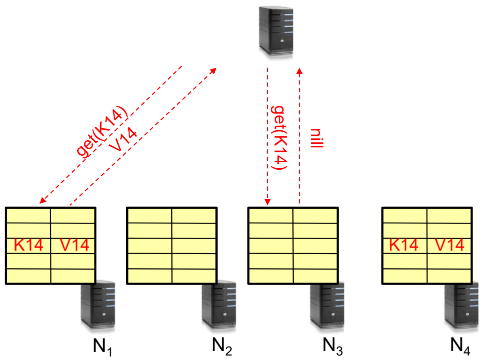
Quorum Consensus Example

- N=3, W=2, R=2
- Replica set for K14: {N1, N2, N4}
- Assume put() on N3 fails



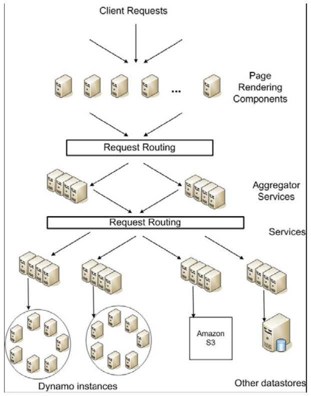
Quorum Consensus Example

- Now, issuing get() to any two nodes out of three will return the answer



DynamoDB Example: Service Level Agreements (SLA)

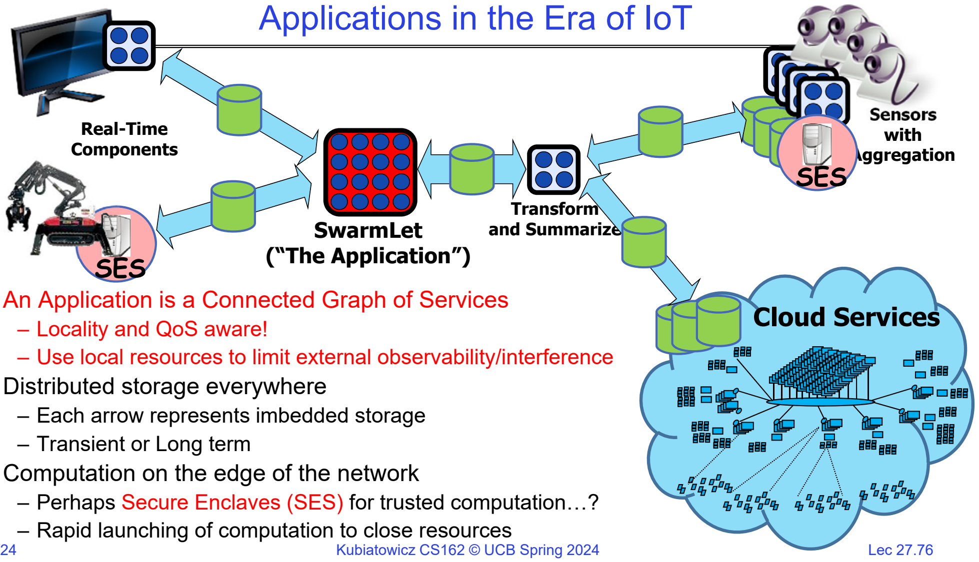
- Dynamo is Amazon's storage system using "Chord" ideas
- Application can deliver its functionality in a bounded time:
  - Every dependency in the platform needs to deliver its functionality with even tighter bounds.
- Example: service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second
- Contrast to services which focus on mean response time



Service-oriented architecture of Amazon's platform

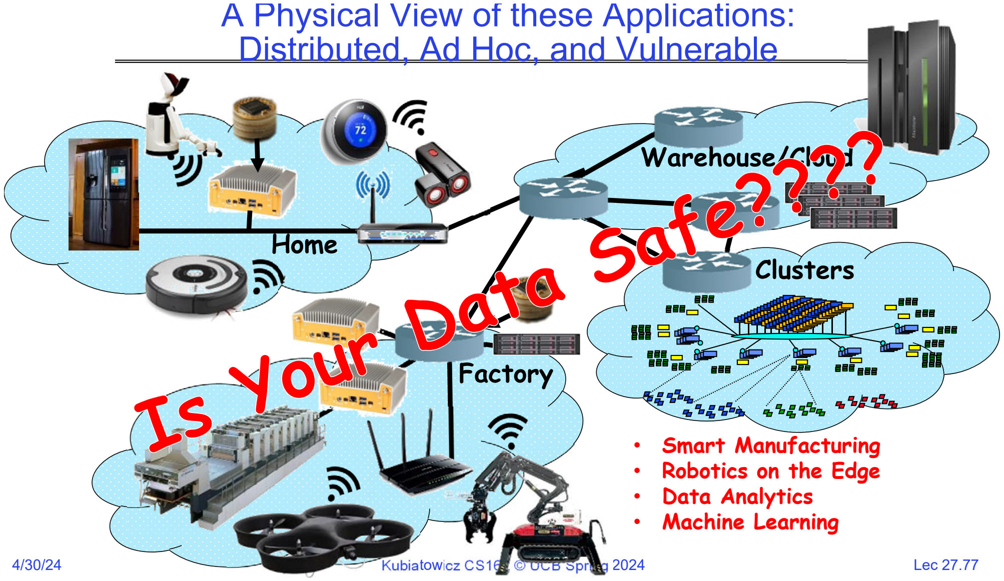
Storage as First Class Citizen: Global Data Plane (GDP)

Applications in the Era of IoT



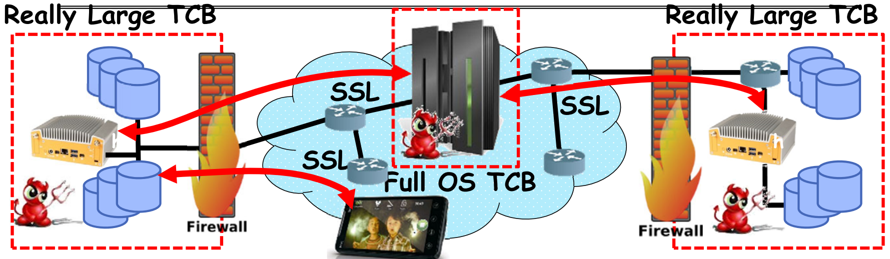
- An Application is a Connected Graph of Services
  - Locality and QoS aware!
  - Use local resources to limit external observability/interference
- Distributed storage everywhere
  - Each arrow represents imbedded storage
  - Transient or Long term
- Computation on the edge of the network
  - Perhaps Secure Enclaves (SES) for trusted computation…?
  - Rapid launching of computation to close resources

A Physical View of these Applications: Distributed, Ad Hoc, and Vulnerable



- **Smart Manufacturing**
- **Robotics on the Edge**
- **Data Analytics**
- **Machine Learning**

Why are Data Breaches so Frequent?



- State of the art: AdHoc boundary construction!
  - Protection mechanisms are all "roll-your-own" and different for each application
  - Use of encrypted channels to "tunnel" across untrusted domains
- Data is protected at the *Border* rather than *Inherently*
  - Large Trusted Computing Base (TCB): huge amount of code must be correct to protect data
  - Make it through the border (firewall, OS, VM, container, etc…) data compromised!
- What about data integrity and provenance?
  - Any bits inserted into "secure" environment get trusted as authentic ⇒ manufacturing faults or human injury or exposure of sensitive information

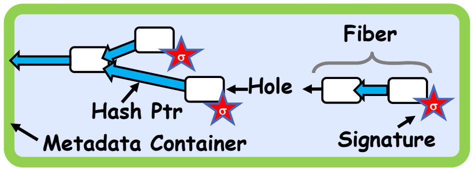
---

On the Importance of Data Integrity

- In July (2015), a team of researchers took *total control* of a Jeep SUV *remotely*
- They exploited a firmware update vulnerability and hijacked the vehicle over the Sprint cellular network
- They could make it speed up, slow down and even veer off the road

- Machine-to-Machine (M2M) communication has reached a dangerous tipping point
  - Cyber Physical Systems use models and behaviors that from elsewhere
  - Firmware, safety protocols, navigation systems, recommendations, …
  - IoT (whatever it is) is everywhere
- Do *you* know where your data came from? *PROVENANCE*
- Do you know that it is ordered properly? *INTEGRITY*
- *The rise of Fake Data!*
  - *Much worse than Fake News…*
  - *Corrupt the data, make the system behave very badly*

The Data-Centric Vision: Cryptographically Hardened Data Containers



- Inspiration: Shipping Containers
  - Invented in 1956. Changed everything!
  - Ships, trains, trucks, cranes handle *standardized format containers*
  - *Each container has a unique ID*
  - *Can ship (and store) anything*
- *Can we use this idea to help?*

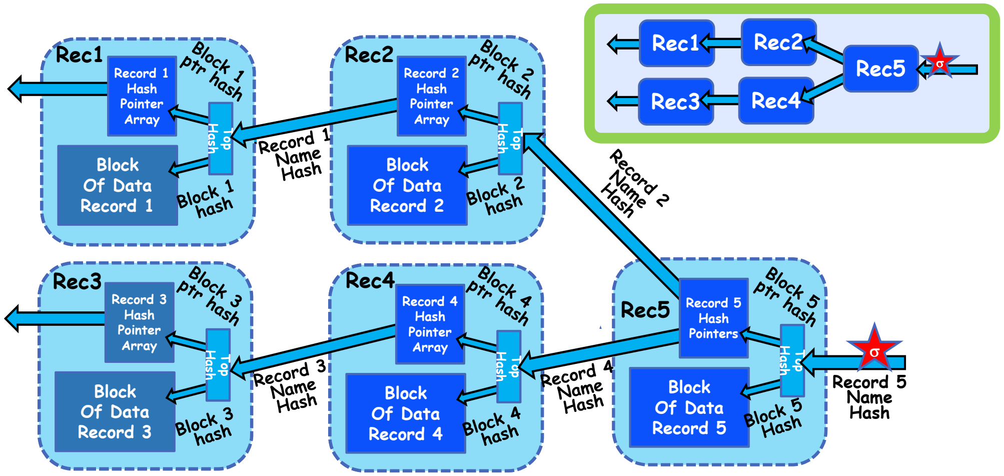
- DataCapsule (DC):
  - Standardized metadata wrapped around opaque data transactions
  - Uniquely named and globally findable
  - Every transaction explicitly sequenced in a hash-chain history
  - Provenance enforced through signatures
- Underlying infrastructure assists and improves performance
  - Anyone can verify validity, membership, and sequencing of transactions (like blockchain)

But – what is a DataCapsule Really?

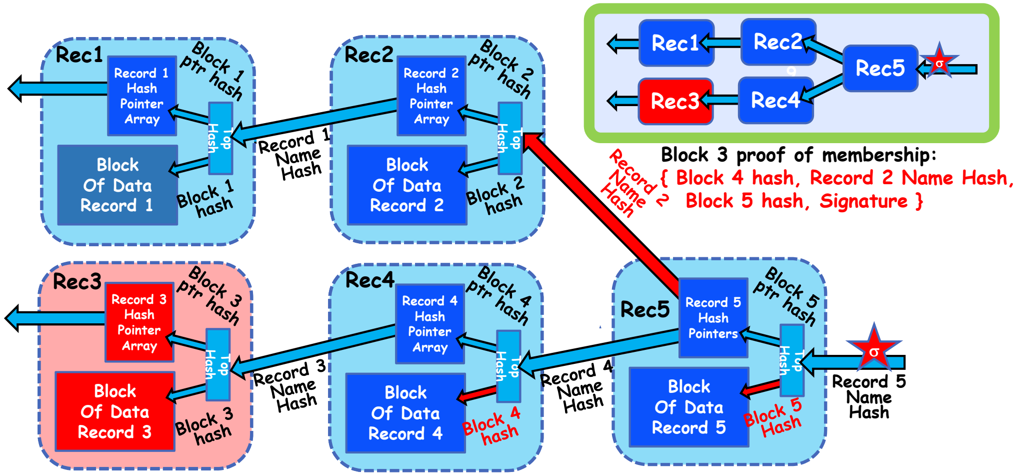
- A cohesive *bundle of data* representing a complete data object:
  - A Key-Value store or a file in a filesystem
  - Any storage model that can be based on a secure log
- A DataCapsule is the *ground truth* of the state of data
  - Everything else is for optimization or durability
- A DataCapsule has a single *owner* which is a cryptographic credential (public/private key pair) that restricts who can write the DataCapsule
  - Writes to the data capsule consist of records signed with the owner key or by key authorized by owner
  - *Records* can represent anything, but must be *linked* to previous records to enforce order
  - *Records* can optionally be encrypted for privacy.
- Reads and writes to a DataCapsule are virtual and over the network
  - *Location-independent, Serverless storage*
  - DataCapsules addressed by name, not location (or IP address)
  - DataCapsule contents signed by owner and encrypted by owner-chosen keys

---

So: DataCapsule is really a “Blockchain in a Box”



DataCapsules provide proof of membership on Reads



Block 3 proof of membership:
{ Block 4 hash, Record 2 Name Hash, Block 5 hash, Signature }

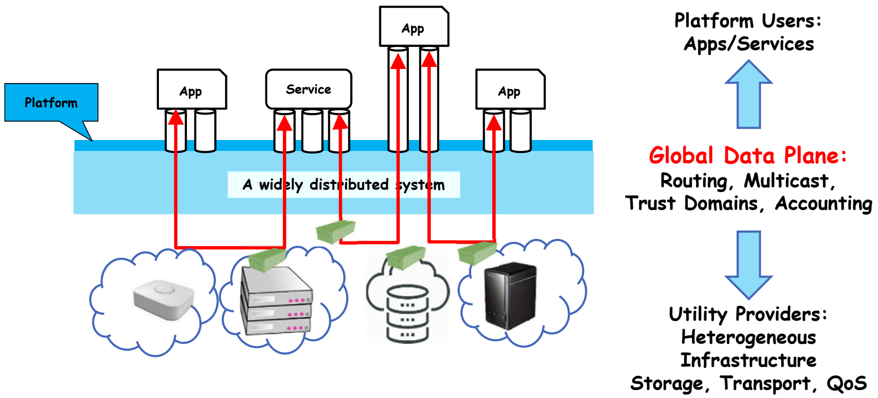
How far can we stretch the shipping container analogy?

- Physical Shipping Containers
  - Shipped over standard transport platforms: planes, trains, trucks, ships
  - Standardized size ⇒ fit on standard transport platforms
  - Standardized labels ⇒ tracking, inventory, routing from one platform to next
  - Contents ⇒ largely unconstrained except for routing constraints (safety, international restrictions, etc…)
- DataCapsules
  - Shipped *and* queried over standard transport platforms: global data plane (GDP) enabled switches with embedded DataCapsule servers and data-centric routing
  - No standardized (maximum) size ⇒ can go anywhere it fits
    - » Instead: standardized metadata ⇒ compatible with any GDP infrastructure
  - Standardized labels ⇒ standard naming of DataCapsules allows for routing of queries from one platform to the next, movement and tracking of actual DataCapsules
  - Contents ⇒ largely unstrained, must adhere to structure requirements (hash-chain structure, signatures) and routing constraints (data safety, international restrictions, etc)

Why does this help?

- The “Networking” effect (Pun Intended!)
  - Standardization ⇒ Infrastructure proliferation that benefits everyone
  - Federation ⇒ Enable a market of service providers
- Data becomes a first-class entity in the network!
  - Asserts its own requirements for security, privacy, which are enforced via cryptography
  - Independent of physical location – policies can target durability, QoS, availability, etc
  - No application silos – data producers own and chose how to share their information
  - Network is informed about the information that it is carrying and where it may go
- First (Necessary) Step:
  Network Cannot Enforce what is not Specified!
- Related information bundled and kept together as it migrates
  - Provenance and data ordering part of all information usage
  - Information labeled with meta-data about (1) Where it is allowed to be within the network, and (2) Who is allowed to view and interact with it, (3) Who is allowed to modify it.

A Platform Approach: the Utility-Provider Model [Ships, Trains, Trucks, and Cranes]

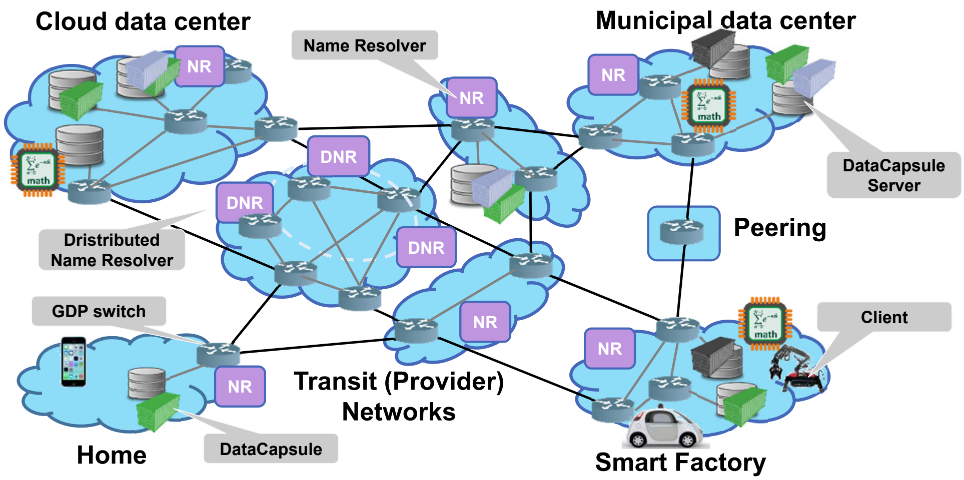


Platform Users: Apps/Services

Global Data Plane:
Routing, Multicast, Trust Domains, Accounting

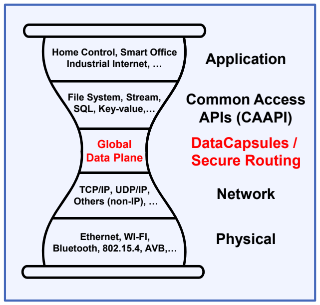
Utility Providers: Heterogeneous Infrastructure Storage, Transport, QoS

A Physical View of the GDP



Refactoring of Applications around Security, Integrity, and Provenance

- Goal: A thin Standardized entity that can be easily adopted and have immediate impact
  - Can be embedded in edge environments
  - Can be exploited in the cloud
  - Natural adjunct to Secure Enclaves for computation
- “Eye-Of-The-Needle” proposition:
  - Thin enough that it will be adopted and enhanced by the most people
  - Powerful enough that application writers can do whatever they need to do
- DataCapsules ⇒ bottom-half of a blockchain?
  - Or a GIT-style version history
  - Simplest mode: a secure log of information
  - Universal unique name ⇒ permanent reference
- Applications writers think in terms of traditional storage access patterns:
  - File Systems, Data Bases, Key-Value stores
  - Called Common Access APIs (CAAPIs)
  - DataCapsules are always the *Ground Truth*



Application

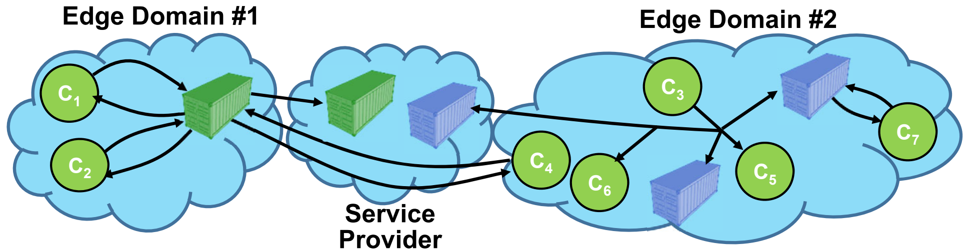
Common Access APIs (CAAPI)

DataCapsules / Secure Routing

Network

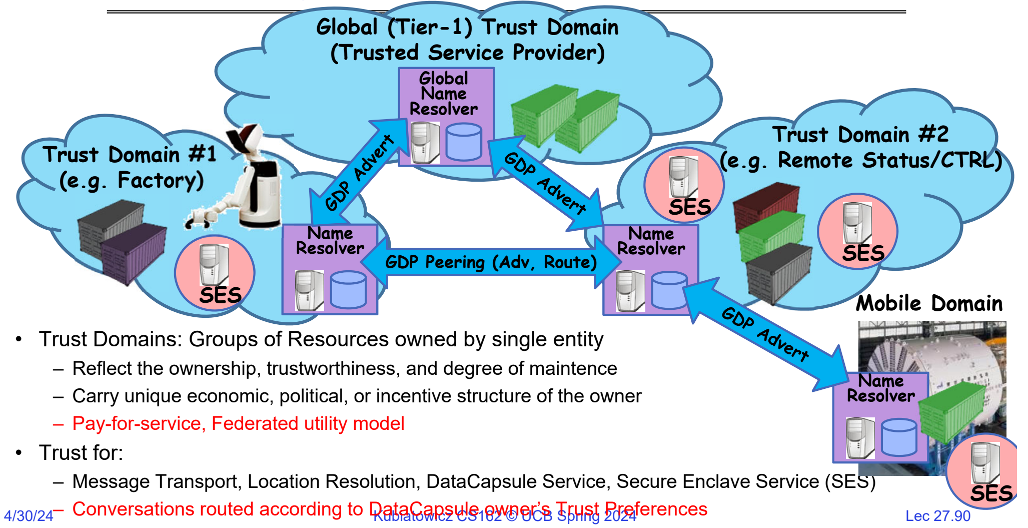
Physical

Global Data Plane (GDP) and the Secure Datagram Routing Protocol



- Flat Address Space Routing
  - Route queries to DCs by names, independent of location (e.g. no IP)
  - DCs move, network deals with it
  - Short-term Channels ("μ-SSL channels")
- Black Hole Elimination: Delegation of Names
  - Only servers authorized by owner of DC may advertise DC service
- Routing only through domains you trust!
  - Secure Delegated Flat Address Routing
- Secure Multicast Protocol
  - Only clients/DC storage servers with proper (delegation) certificates may join
- Queries (messages) are *Fibers*
  - Self-verifying chunks of DataCapsules
  - Writes include appropriate credentials
  - Reads include proofs of membership
- Incremental deployment as an overlay
  - Prototype tunneling protocol ("GDPinUDP")
  - Federated infrastructure w/routing certificates

Reasoning about the infrastructure: Trust Domains



- Trust Domains: Groups of Resources owned by single entity
  - Reflect the ownership, trustworthiness, and degree of maintence
  - Carry unique economic, political, or incentive structure of the owner
  - Pay-for-service, Federated utility model
- Trust for:
  - Message Transport, Location Resolution, DataCapsule Service, Secure Enclave Service (SES)
  - Conversations routed according to DataCapsule owner's Trust Preferences

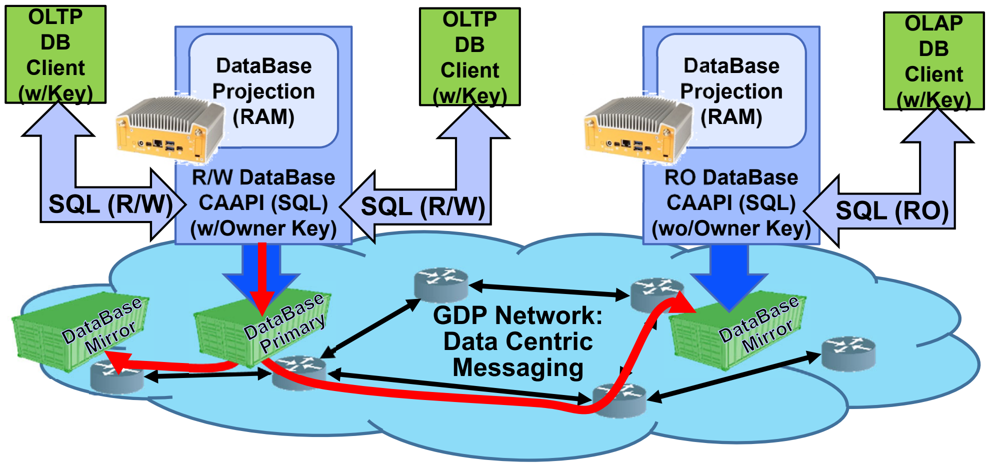
---

Common Access APIs (CAAPIs)

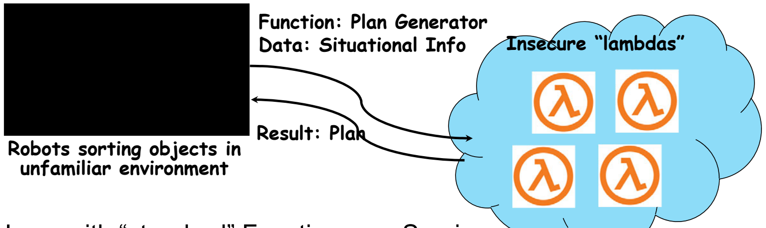
- Common Access APIs (CAAPIs) provide convenient/familiar *Storage Access Patterns*:
  - Random File access, Indexing, SQL queries, Latest value for given Key, etc
  - Optional Checkpoints for quick restart/cloning
  - Refactoring: CAAPIs are services or libraries running in trusted or secured computing environments on top of DataCapsule infrastructure
- Many Consistency Models possible
  - DataCapsules are "Conflict-free Replicated Data Types" (CRDTs): Synchronization via Union
  - Single-Writer CAAPIs prevent branches if sufficient stable storage (strong consistency models)
  - DataCapsules with branches: like GIT or Amazon Dynamo (write always, reader handles branches)
  - CAAPIs can support anything from weak consistency to serializability
- Examples:
  - Streaming storage
  - Key/Value store with time-travel
  - Filesystem (changeable sequences of bytes organized in hierarchy)
  - Multi-writer storage using Paxos or RAFT
  - Byzantine agreement with threshold admission to DataCapsules

---

Example #1: Using DataCapsules to build more sophisticated data access patterns (e.g. DataBase)



Example #2: Function as a Service

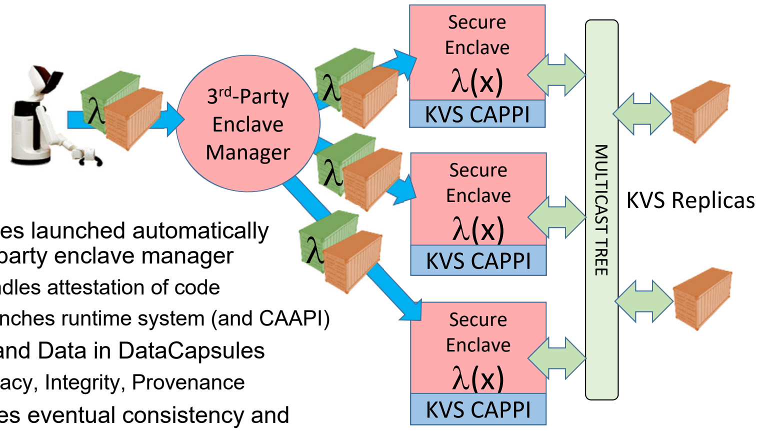


- Problems with "standard" Function as a Service:
  - Runs entirely in uncontrolled cloud
  - No explicit state: must add something else
  - Not private, leaks algorithms *and* results
- Opportunity cost: Can't exploit Edge!
  - Latency, predictability, privacy, ….

Function-as-a-Service(Faas) Cloud Service Provider ("lambdas")

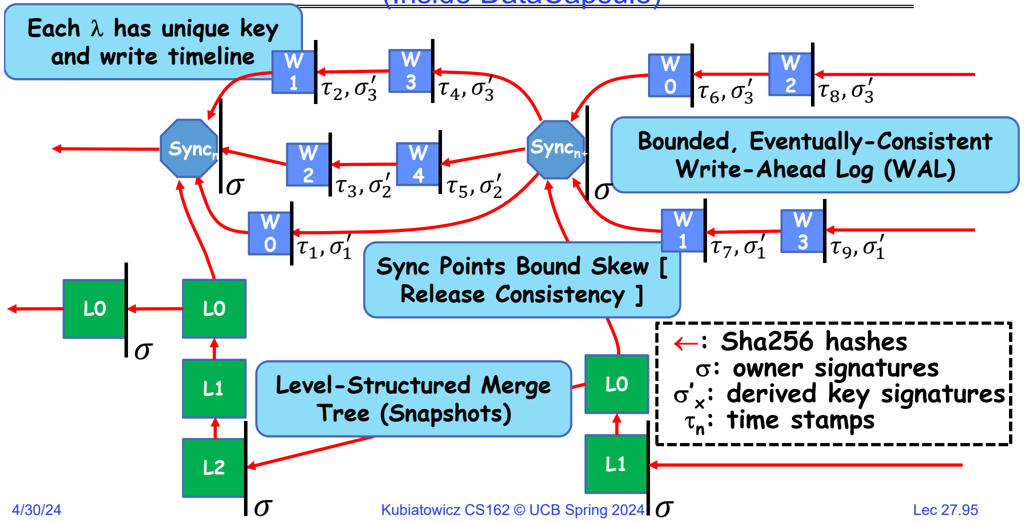
AWS Lambda Google Cloud

A better Alternative: Paranoid Stateful Lambdas (PSL)



- Enclaves launched automatically by 3rd-party enclave manager
  - Handles attestation of code
  - Launches runtime system (and CAAPI)
- Code and Data in DataCapsules
  - Privacy, Integrity, Provenance
- Provides eventual consistency and release consistency

Multi-writer Model for Parallel Key-Value Store: (Inside DataCapsule)



Each λ has unique key and write timeline

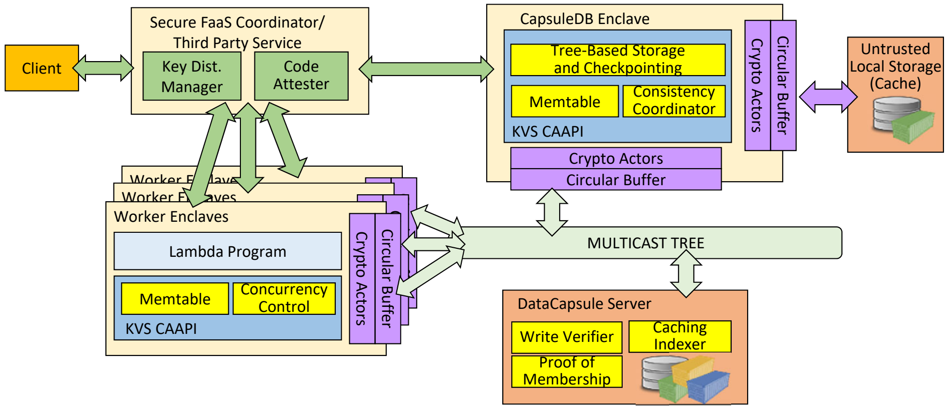
Bounded, Eventually-Consistent Write-Ahead Log (WAL)

Sync Points Bound Skew [Release Consistency]

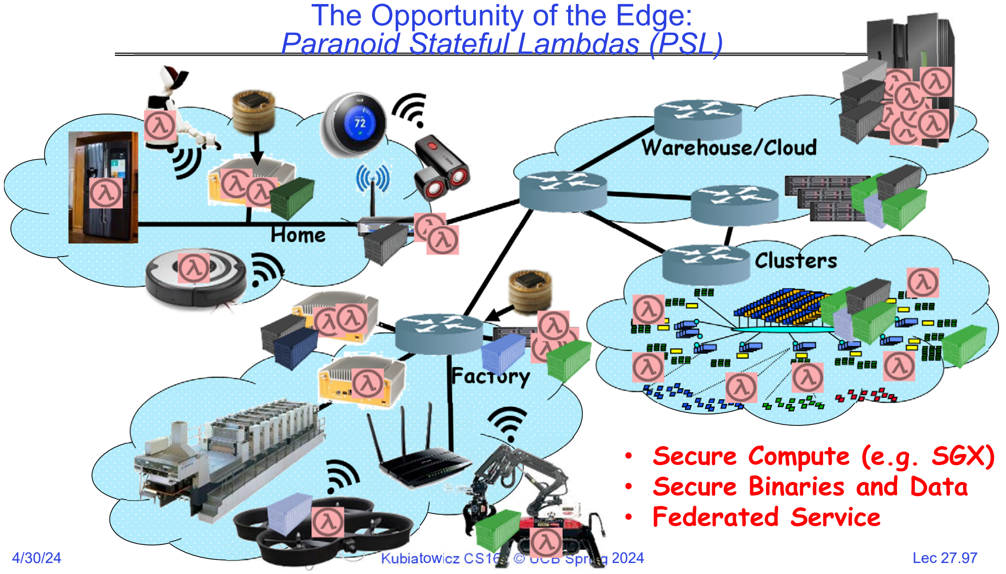
Level-Structured Merge Tree (Snapshots)

←: Sha256 hashes
σ: owner signatures
σ'_x: derived key signatures
τ_n: time stamps

Paranoid Stateful Lambdas: Key-Value Store CAAPI for Secure FaaS

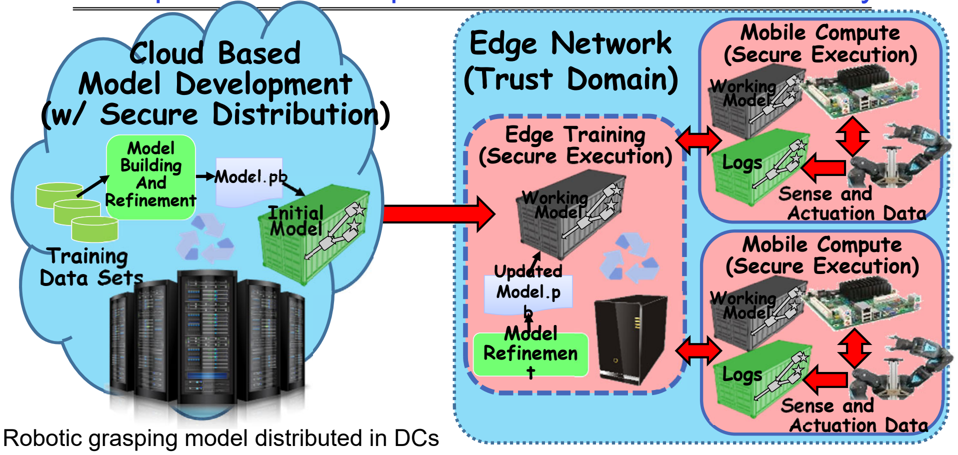


The Opportunity of the Edge: *Paranoid Stateful Lambdas (PSL)*



- ***Secure Compute (e.g. SGX)***
- ***Secure Binaries and Data***
- ***Federated Service***

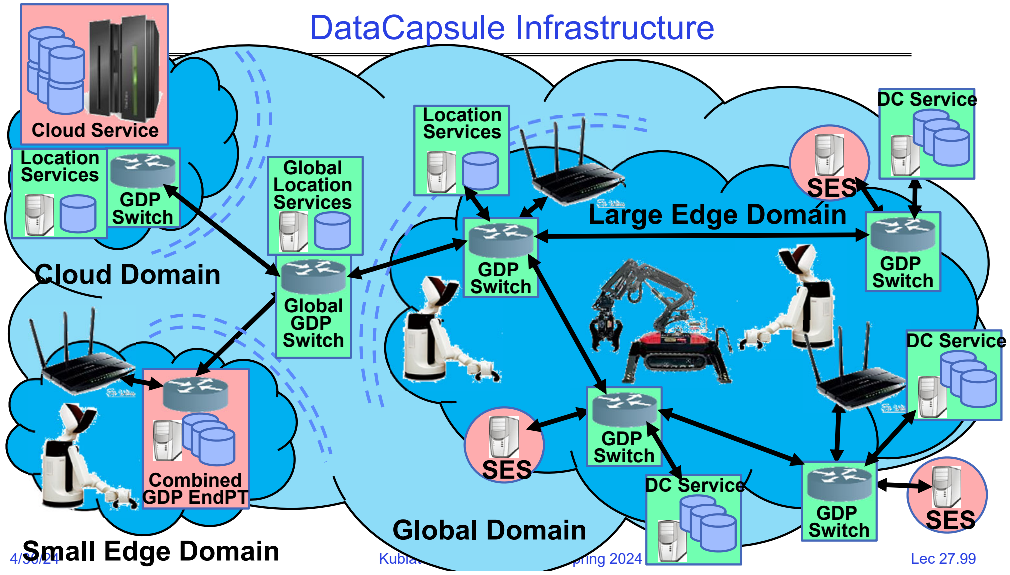
Example #3: Data Capsules as Part of Model Delivery



- Robotic grasping model distributed in DCs
  - Intellectual property of producer (only unpacked in environments guaranteed not to leak model)
  - Refinement on the edge is updated only by authorized enclaves with attested algorithms

---

DataCapsule Infrastructure

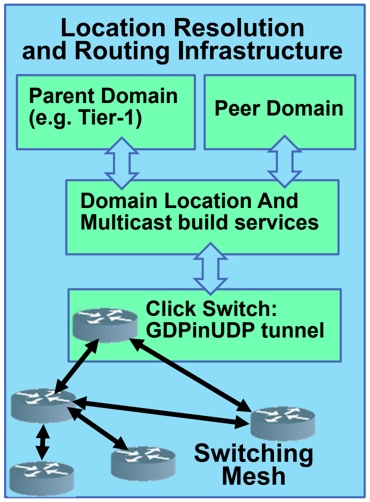


Research Agenda: What is Hard?

- Biggest Challenge: Convince People to Refactor their applications around DataCapsules
  - Incremental Deployment encouraged via (1) overlay networking followed by (2) “native” GDP datagram routing – possibly even without IP service
  - CAAPIs provide standardized storage “patterns” for naïve and domain application writers
- DataCapsules provide extremely flexible storage (intended as a primitive element upon which to build a wide array of storage systems)
  - The trick is to provide understandable semantics with good performance
  - Consider wide range of Google storage systems (GFS, BigTable, Megastore, Spanner…)!
- DataCapsule placement: Edge vs Cloud
  - Placement based on Performance, Privacy Constraints, Durability Requirements, BW, QoS, ….
- Replication and Failover semantics
  - Basic Replication simple since DataCapsules are CRDTs (Conflict-Free Replicated Datatypes). Thus, synchronization is via union of DataCapsules is easy
  - Providing quick adaptation in (routing) network as DataCapsule servers fail and recover while still providing understandable semantics is tricky
- Replication in the presence of network partitions and malicious agents
  - Can provide multi-writer storage using Paxos or RAFT
  - Can use Byzantine agreement with threshold admission to DataCapsules

Research Agenda (con't): What is Hard?

- Flat Address Space Routing is Dead, long live Flat Address Space Routing
  - No physical hierarchy in the names of DataCapsules
  - Each advertising certificate (Delegated Flat Name) is unforgeable (RO) and easily exported using a scalable DHT
  - Using Redis key-value store for initial prototype
- Adaptable, Authenticated, Automatic Multicast construction
  - Multicast is an old topic, but secure, performant, multicast that respects trust domains is essential to DataCapsule/GDP
  - Can leverage ideas from prior Bayeux multicast DHT work
- Only Active Conversations Stored in Switches!
  - Provides hope of scalability, but challenge of routing
- QoS-Aware Routing problem: Efficiently routing while respecting QoS and exploiting hardware (e.g., TSN)
  - Can leverage ideas from prior Brocade landmark overlay DHT work



Location Resolution and Routing Infrastructure

Parent Domain (e.g. Tier-1)

Peer Domain

Domain Location And Multicast build services

Click Switch: GDPinUDP tunnel

Switching Mesh

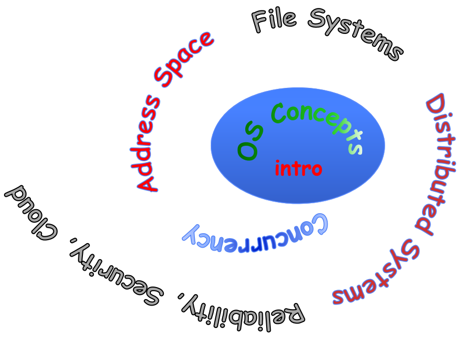
Why the Global Data Plane Again???

- Yes, you *could:*
  - *Provide your own infrastructure for everything*
  - *Provide your own storage servers*
  - *Provide your own networking, location resolvers, intermediate rendezvous points*
- But: *Why?*
  - Standardization is what made the IP infrastructure so powerful
  - Utilize 3rd-party infrastructure owned (and constantly improved) by others
  - Sharing is much harder with stovepiped solutions!
- The Global Data Plane provides *standardized infrastructure support*
  - It provides a standardized substrate for secure flat routing and publish-subscribe multicast
  - It provides a provides the ability to reason about infrastructure providers (Trust Domains)
  - It frees DataCapsules from being tied to a particular physical location
  - ⇒ Analogous to ships, planes, trains, and cranes that support shipping containers
- The GDP routes conversations between endpoints such as DataCapsules, sensors, actuators, services, clients, etc.
- Information protected in DataCapsules, but freed from physical limitations by the GDP
  - Correctness and Provenance *enforced* by DataCapsules
  - Performance, QoS, and Delegation of Trust handled by the GDP

GDP: Conclusion

- The most game-changing element of this agenda is the presence of ubiquitous, secure and mobile bundles of data: DataCapsules
  - Provably authentic and self-consistent
  - Only authorized writers can add information; anyone with possession can verify integrity
- The power of DataCapsules are in *standardization*
  - If everyone uses DataCapsules, then everyone reaps the benefits—
No malicious information, no fake news, no breached passwords
  - Eliminate rampant "roll-your-own" philosophy that yields data breaches
- Naturally Coupled with Secure Edge Computing (Enclaves)
- Burden of standardization reduced through careful design:
  - Incremental, flat-address-space routing (no IP addresses!)
  - Efficient refactoring of communication around storage
  - Familiar storage patterns (facades): File Systems, DataBases, Key-Value Stores, Streams,…
- Exciting new applications: Robotics and Machine Learning

Thank you!



- Thanks for all your great questions!
- Good Bye! You have all been great!