Distributed systems
Lecture 16: Security and NASD/AFS/Coda case studies

Dr Robert N. M. Watson

1

Last time

- Looked at replication in distributed systems
- **Strong consistency**:
  - Approximately as if only one copy of object
  - Requires considerable coordination on updates
  - Transactional consistency & **quorum systems**
- **Weak consistency**:
  - Allow clients to potentially read stale values
  - Some guarantees can be provided (FIFO, eventual, session), but at additional cost to availability
- Amazon/Google case studies: Dynamo, MapReduce, BigTable, Spanner.

2

Distributed-system security

- Distributed systems span **administrative domains**
- Natural to extend **authentication**, **access control**, **audit**, to distributed system, but can we:
  - Distribute local notions of a **user** over many machines?
  - Enforce system-wide properties – e.g., **personal data privacy**?
  - Allow systems operated by parties to **interact safely**?
  - Not require that networks be safe from **monitoring/tampering**?
  - **Tolerate compromise** a subset of nodes in the system?
  - Provide **reliable service** to most users even under attack?
  - Accept and tolerate **nation-state actors** as adversaries?
- For a system to offer secure services, it must be secure
  - **Trusted Computing Base** (**TCB**) – minimum software (or hardware) required for a system to be secure
  - Deploy compartmentalization-style sandboxing structures

3

Access control

- Distributed systems may want to allow access to resources based on a security policy
- As with local systems, three key concepts:
  - **Identification**: who you are (e.g. user name)
  - **Authentication**: proving who you are (e.g. password)
  - **Authorization**: determining what you can do
- Can consider authority to cover actions an authenticated subject may perform on objects
  - **Access Matrix** = set of rows, one per **subject**, where each column holds allowed operations on some **object**

4

The access-control matrix

	Object$_1$	Object$_2$	Object$_3$	…
User$_1$		+read		
User$_2$	+read +write	+read		
Group$_1$	-read		+read +write	
…				

- **A(i, j)**
  - Rows represent principals (sometimes groups)
  - Columns represent objects
  - **Cell(i, j)** contain access rights of row **i** on object **j**
- Access matrix is typically large & sparse:
  - Just keep non-NULL entries by column or by row
- Tricky questions
  - How do you name/authenticate users, and who can administer groups?
  - How do you compose conflicting access-control rules (e.g., **user1 +read** but **group1 –read**)?
  - What consistency properties do access control, groups, and users require?

5

Access Control Lists (ACLs)

- Keep columns: for each object, keep **list of subjects** and **allowable access**
  - ACLs stored with objects (e.g. local filesystem)
  - Key primitives: **get/set**
  - Like a guest list on the door of a night club
- ACL change should (arguably) immediately grant/deny further access
  - What does this mean for distributed systems?
  - Or even local systems (e.g., UNIX)

6

Capabilities

- Capabilities are **unforgeable tokens of authority**
  - Keep rows: for each subject **S**, keep list of objects / allowable accesses
  - Capabilities stored with subjects (e.g. processes)
  - Bit like a key or access card that you carry around
  - Think of as **secure references** – if you hold a reference to an object, you can use the object
- Key primitive: **delegation**
  - Client can delegate capabilities it holds to other clients (or servers) in the system to act on its behalf
  - Downside: revocation may now be more complex

7

Access control in distributed systems

- Single systems often have small number of users (**subjects**) and large number of **objects**:
  - E.g. a hundred of users in a Unix system
  - Track subjects (e.g. users) and store ACLs with objects (e.g. files)
- Distributed systems are large & dynamic:
  - Can have huge (and unknown?) number of users
  - Interactions via network; no explicit 'log in' or user processes
- Capability model is a more natural fit:
  - Client presents capability with request for operation
  - System only performs operation if capability checks out
  - Avoid synchronous RPCs to check identities/policies
- Not mutually exclusive: ACLs can grant capabilities
- Can't trust nodes/links: use **cryptography with secret keys**
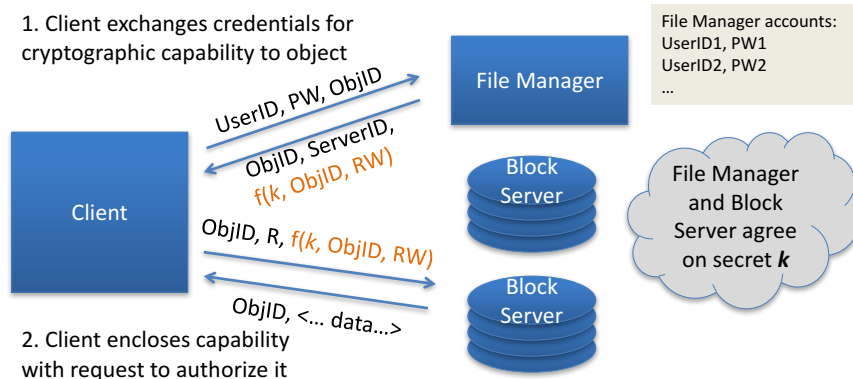
8

Cryptographic capabilities

- How can we make capabilities **unforgeable**?
- Capability server could issue capabilities
  - User presents credentials (e.g., username, password) and requests capabilities representing specific rights
  - E.g. capability server has secret key **k** and a one-way function **f**()
  - Issues a capability <*ObjID*, *access*, **f**(**k**, *ObjID*, *access*) >
  - Simple example is **f**(k,o,a) = **SHA256**(k|o|a)
- Client transmits capability with request
  - If object server knows **k**, can check operation
- Can use same capability to access many servers
  - And one server can use it on your behalf (e.g., web tier can request objects from storage tier on user's behalf)
- More mature scheme might use public key crypto (why?)

9

Distributed capability example: NASD

1. Client exchanges credentials for cryptographic capability to object

File Manager accounts:
UserID1, PW1
UserID2, PW2
…

UserID, PW, ObjID

File Manager

ObjID, ServerID,
f(k, ObjID, RW)

Client

Block Server

File Manager and Block Server agree on secret *k*

ObjID, R, f(k, ObjID, RW)

ObjID, <… data…>

Block Server

2. Client encloses capability with request to authorize it

- **Network-Attached Secure Disks (NASD)** – Gibson, et al 1997 (CMU)
- Clients access remote disks directly rather than via through servers
- "File Manager" grants client systems **capabilities** delegating direct access to objects on network-attached disks – as directed by ACLs

10

Capabilities: pros and cons

- Relatively simple and pretty scalable
- Allow anonymous access (i.e. server does not need to know identity of client)
  - And hence easily **allows delegation**
- However this also means:
  - Capabilities can be stolen (unauthorized users)…
  - … and are **difficult to revoke** (like someone cutting a copy of your house key)
- Can address these problems by:
  - Having time-limited validity (e.g. 30 seconds)
  - Incorporating version into capability, store version with the object: increasing version => revoke all access

11

Combining ACLs and capabilities

- Recall one problem with ACLs was inability to scale to large number of users (subjects)
- However in practice we may have a small-ish number of authority levels
  - E.g. moderator versus contributor on chat site
- **Role-Based Access Control (RBAC):**
  - Have (small-ish) well-defined number of **roles**
  - Store ACLs at objects based on roles
  - Allow subjects to **enter** roles according to some rules
  - Issue capabilities which attest to current role

12

Role-based access control (RBAC)

- General idea is very powerful
  - Separates { **principal → role** }, { **role → privilege** }
  - Developers of individual services only need to focus on the rights associated with a role
  - Easily handles evolution (e.g. an individual moves from being an undergraduate to an alumnus)
- Possible to have sophisticated rules for role entry:
  - E.g. enter different role according to time of day
  - Or entire role hierarchy (1B student <= CST student)
  - Or parametric/complex roles ("the doctor who is currently treating you")

13

Single-system sign on

- Distributed systems involve many machines
  - Frustrating to have to authenticate to each one!
- Single-system sign-on: security with lower user burden
  - E.g. Kerberos, Microsoft Active Directory let you authenticate to a single **domain controller**
  - Bootstrap via password/private key + cert. on smart card
  - Get a **session key** and a **ticket** (~= a capability)
  - Ticket is for access to the **ticket-granting server** (**TGS**)
  - When wish to e.g. log on to another machine, or access a remote volume, s/w asks **TGS** for a ticket for that resource
  - Notice: **principals** might could be users … or **services**
- Other wide-area "federated" schemes
  - Multi-realm Kerberos, OpenID, Shibboleth

14

AFS and Coda

- Two 1990s CMU distributed file systems that helped create our understanding of distributed-system scalability, security, …
  - **AFS**: Andrew File System "campus-wide" scalability
  - **Coda**: Add write replication, weakly connected or fully disconnected operation for mobile clients
- Scale distributed file systems to **global scale** using a concurrent and distributed-system ideas
  - Developed due to NFS scalability failures
  - RPC, close-to-open semantics, pure and impure names, explicit cache management, security, version vectors, optimistic concurrency, quorums, multicast, …
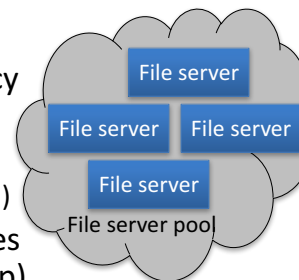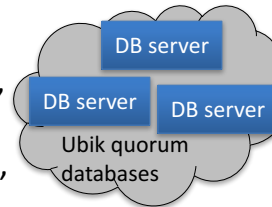
15

The Andrew File System (AFS)

- Carnegie Mellon University (1980s) address performance, scalability, security weaknesses of NFS
- Global-scale distributed filesystem
  - **/afs/cs.cmu.edu/user/rnw/public_html/index.html, /afs/ibm.com/public**
    - **Cells** incorporate dozens or hundreds of servers
  - Clients transparently merge namespaces and hide file replication/migration effects
  - Authentication/access control w/Kerberos, group servers
  - Cryptographic protection of all communications
  - Mature non-POSIX semantics (**close-to-open**, **ACLs**)
- Still in use today; open sourced as OpenAFS
- Inspired **Distributed Computing Environment (DCE)**, **Microsoft's Distributed File System (DFS)**, and **NFSv4**
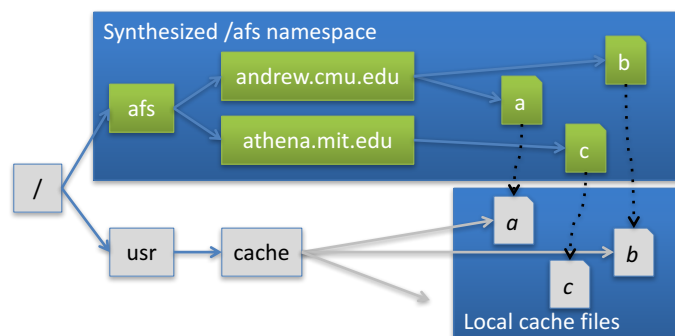
16

AFS3 per-cell architecture

- **Client-server** and **server-server RPC**
- **Ubik** quorum database for authentication, volume location, and group membership
- Namespace partitioned into volumes; e.g., /**afs**/**cmu.edu**/**user**/**rnw**/public_html traverses four volumes
- Unique **ViceID**s: **{CellID, VolumeID, FID}**
- Volume servers allow limited redundancy or higher-performance bulk file I/O:
  - **read-write on a single server** (~rnw)
  - **read-only replicas on multiple servers** (/bin)
- Inter-server **snapshotting** allows volumes to migrate transparently (with client help)

DB server
DB server   DB server
Ubik quorum databases

File server
File server   File server
File server
File server pool

17

Persistent client-side caching in AFS

Synthesized /afs namespace

afs
andrew.cmu.edu
athena.mit.edu
a
b
c

/
usr   cache
a
b
c
Local cache files

- AFS implements **persistent caches** on client-side disks
- Vnode operations on remote files are redirected to local **container files** for local I/O performance
- Non-POSIX **close-to-open semantics** allow writes to be sent to the server only on **close**()

18

AFS callback promises
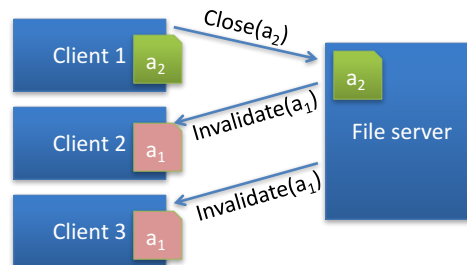
- Servers issue **callback promises** on files held in client caches
- When a file server receives a write-**close**() from one client, it issues **callbacks** to invalidate copies in other client caches
- Unlike NFS, no synchronous RPC is required when opening a cached file: if callback is not been broken, cache is fresh
- However, client write-**close**() is **synchronous**: can't return until callbacks acknowledged by other clients – why?

19

The Coda File System

- Developed at Carnegie Mellon University in the 1990s
    - Starting point: open-sourced AFS2 from IBM
- Improve **availability**: optimistic replication, offline mode:
    - Improve availability through **read-write replication**
    - Improve performance for **weakly connected clients**
    - Support mobile (sometimes) **fully disconnected clients**
- Exploit network features to improve performance:
    - **Multicast RPC** to efficiently send RPCs to groups of servers
- Exchange **weaker consistency** for **stronger availability**
    - **Version vectors** for directories, files identify write conflicts
    - **Users resolve some conflicts** … with (very) mixed results?
- Surprising result: unplug network to make builds go faster
    - It is faster to journal changes to local disk (offline) and reconcile later than synchronously write to distributed filesystem (online)

20

Summary (1)

- Distributed systems are everywhere
- Core problems include:
  - Inherently concurrent systems
  - Any machine can fail…
  - … as can the network (or parts of it)
  - And we have no notion of global time
- Despite this, we can build systems that work
  - Basic interactions are request-response
  - Can build synchronous RPC/RMI on top of this …
  - Or asynchronous message queues or pub/sub

21

Summary (2)

- Coordinating actions of larger sets of computers requires higher-level abstractions
  - Process groups and ordered multicast
  - Consensus protocols, and
  - Replication and Consistency
- Various middleware packages (e.g. CORBA, EJB) provide implementations of many of these:
  - But worth knowing what's going on "under the hood"
- Recent trends towards even higher-level:
  - **MapReduce** and friends

22