Distributed systems

Lecture 8: Security; AFS/Coda

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
- **Service replication**:
  - **Stateless** (easy!) or **Passive** (primary/backup) common, **Active** (state-machine replication) less so
- Google datacenter case studies: MapReduce, BigTable, etc.

2

Distributed-system security

- Distributed systems span administrative domains; content from many users and organizations
- It seems 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 such as 'personal data privacy'?
  - Allow systems operated by different 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 when under attack?
  - Accept and tolerate nation-state actors as adversaries?
- Very hard problems – but we can't build truly scalable distributed systems without trying to solve them!

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

ACLs and capabilities

- Access matrix is typically large & sparse:
  - Just keep non-NULL entries by column or by row
- **Access Control Lists**:
  - Keep columns: for each object, keep list of subjects / allowable access
  - ACLs stored with objects (e.g. local filesystem)
  - Like a guest list on the door of a night club
- **Capabilities**:
  - 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
- Not mutually exclusive: ACLs as a policy for granting capabilities
  - E.g., UNIX permissions are checked on open(), not on read(), write()
  - But observe effect on revocation: changing permissions does not revoke outstanding capabilities

5

Access control in distributed systems

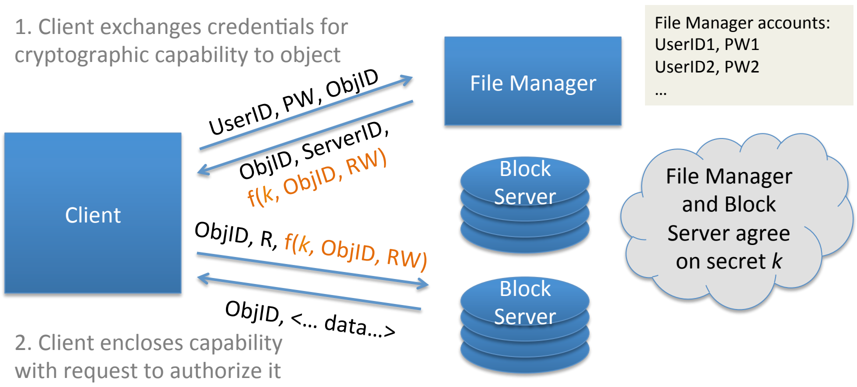
- Single systems often have small number of users (subjects) and large number of objects:
  - e.g. a few hundred users in a Unix system
  - Track subjects (e.g. user IDs) and store ACLs with objects (e.g. files)
- Distributed systems are large & dynamic:
  - Can have huge (and unknown?) number of users
  - Interactions via network – no explicit ‘log in’ or per-user process
- 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/access-control policies
- Can’t trust nodes or links: rely on **cryptography with secret keys**

6

Cryptographic Capabilities

- Capabilities are **unforgeable tokens of authority**
- 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?)

Distributed capability example: NASD



- Network-Attached Secure Disks (NASD) – Gibson, et al 1997 (CMU); actual protocol somewhat more complicated than this example
- Improve network file system scalability by allowing clients to directly access remote disks rather than indirecting through servers
- "File Manager" grants client systems capabilities delegating direct access to objects on network-attached disks

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, and storing version with the object: increasing version => revoke all access

9

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

10

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")

11

Single-system sign on

- Distributed systems involve many machines
  - Frustrating to have to authenticate to each one!
- Single-system sign on aims to ease user burden while maintaining good security
  - e.g. Kerberos, Microsoft Active Directory let you authenticate to a single **domain controller**
  - Bootstrap using a password or private key / certificate 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
  - Schemes
  - Notice: **principals** might could be users … or even services
- Some wide-area "federated" schemes too (Multi-realm Kerberos, OpenID, Shibboleth)

12

Note: not covered in lecture

AFS and Coda

- Two CMU distributed file systems that helped create our understanding of distributed-system scalability
  - **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 broad and mature set of concurrent and distributed-system ideas
- RPC, close-to-open semantics, pure and impure names, explicit cache management, security, version vectors, optimistic concurrency, multicast, journaling, …

13

Note: not covered in lecture

The Andrew File System (AFS)

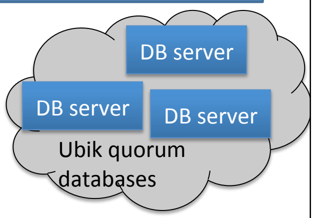
- Carnegie Mellon University (1980s) address performance, scalability, security weaknesses of NFS
- Global-scale distributed filesystem
  - /afs/cs.cmu.edu/user/rnw, /afs/ibm.com/public
  - **Cells** transparently incorporate dozens or hundreds of servers
  - Clients merge namespaces and hide replication/migration of files
  - Distributed authentication/access control w/Kerberos, group servers
  - (Optional) cryptographic protection of all communications
  - Quorum-backed metadata databases for UserDB, VolDB, etc.
  - Persistent client caches, servers aware of client cache contents
  - Mature non-POSIX filesystem semantics (close-to-open, ACLs)
- Still in use at large institutions today; open sourced as OpenAFS
- Inspiration many aspects of Distributed Computing Environment (DCE) and Microsoft's Distributed File System (DFS)

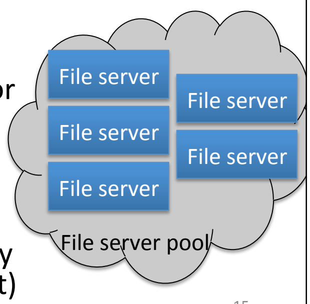
14

Note: not covered in lecture

AFS3 per-cell architecture

- **Client-server** and **server-server** communication via 'rx' RPC package
- **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
- Special symlinks provide volume linkage
- Files ID'd by **ViceID**: {CellID, VolumeID, FID}
- Volume servers trade limited redundancy for higher-performance bulk file I/O:
  - **read-write on a single server** (~rnw)
  - **read-only replicas on multiple servers** (/bin)
- Efficient inter-server snapshot algorithm allows volumes to be migrated transparently while in use by users (with help of AFS client)

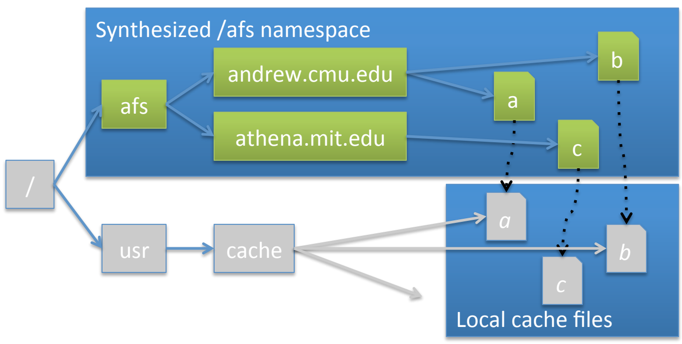




15

Note: not covered in lecture

Persistent client-side caching in AFS

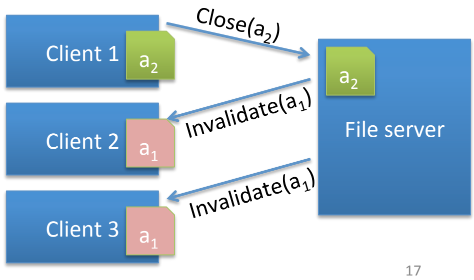


- AFS implements **persistent caches** on client-side disks
- Vnode operations on remote files are redirected to local **container files** for local I/O performance
- **Close-to-open semantics** allow writes to be sent to the server only on close()

16

Note: not covered in lecture

AFS callback promises



- AFS servers issue **callback promises** on files held in client caches
- When a file server receives a write-close() from one client, it initiates **callbacks** to invalidate cached copies on other clients
- Unlike NFS, no synchronous RPC is required when opening a cached file: the callback has not been broken so it must be fresh
- However, client write-close() is synchronous: can't return until callbacks acknowledged by other clients – why?
- What consistency properties might we want for ACLs?

17

Note: not covered in lecture

The Coda File System

- Developed at Carnegie Mellon University in the 1990s by M. Satyanaraynan's group
- Starting point: open-sourced AFS2 from IBM
- Improve **availability** through optimistic replication and client-side caching/journaling:
  - Improve availability through **read-write replication**
  - Improve performance for **weakly connected clients**
  - Support mobile (sometimes) **fully disconnected clients**
- Exploit new network features to improve performance:
  - Multicast RPC to efficiently send RPCs to groups of servers
- Key design challenge: trade off exposing weak consistency to user in return for availability

18

Note: not covered in lecture

Coda read-write replication

- Volumes (hence files) are stored on **Volume Storage Groups (VSGs)** rather than on a single volume server as in AFS
- Coda associates a **version vector** with each file
  - Like a vector clock only per-object rather than per process
  - Each vector entry corresponds to one VSG server's version of the file
- Reachable VSG subset is the **Accessible Volume Storage Group (AVSG)**
- Clients read from any server, multicast writes to all: **read-one, write-all**
  - When fully online (AVSG = VSG) , close() is synchronous; writes ordered
  - On partition/server outage (AVSG $\subset$ VSG), writes are still permitted
  - As servers recover, client access triggers **server-server resolution**
  - If version vectors allow causal order to be established, resolution is automatic
  - Most non-causal server-server directory conflicts can be automatically resolved (why?)
  - For files, **user-directed** or **application-specific conflict resolution** is required
- What if a user is asked to resolve a conflict on a file they didn't modify?

19

Note: not covered in lecture

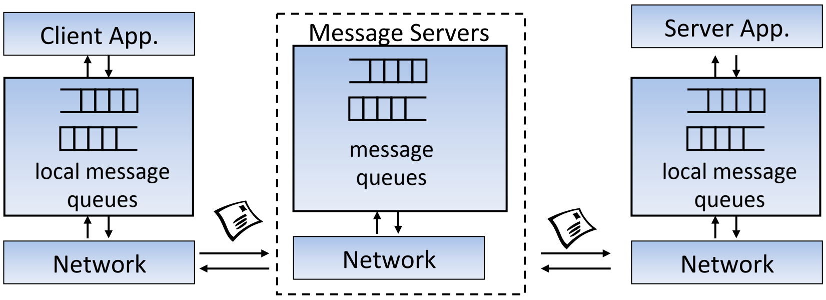
Coda disconnected operation

- Mid-1990s, mobile computing was becoming available for the first time – devices often had weak or no connectivity
- Coda allows mobile-client operations to continue against the persistent cache even when operating disconnected (AVSG = $\varnothing$)
- **Hoarding**: prior to going offline, users can provide Coda with policy as to which files should be preemptively loaded into the cache (e.g., user ~)
- Offline writes are logged in the **Client Modification Log (CML)**
  - When going back online, CML is replayed against AVSG (**reintegration)**
  - **CML optimization** deletes NOP sequences: e.g., create+delete a temp file
  - Client-server conflicts, as with server-server, are detected via version vectors
  - User/application must handle conflicts that can't be resolved automatically
  - Is this better than the server-server conflict resolution case?
- Curious: if Ethernet unplugged, my build goes faster – why?
  - Clever trick for weakly connected clients: if network is bottleneck, take volume offline and log changes, trickling them back asynchronously until caught up
- These ideas have influenced systems like Microsoft's "offline folders"

20

Note: not covered in lecture

Coordination Services



- Earlier looked at middleware support for RPC/RMI
  - Imperative and (typically) synchronous interaction
- An alternative is **message-oriented middleware**
  - Communication via asynchronous messages
  - Messages stored in **message queues**

21

Note: not covered in lecture

MOM: Pros and Cons

- **Asynchronous interaction**
  - Client and server are only loosely coupled
  - Messages are queued
  - Good for application integration
- Support for **reliable delivery service**
  - Keep queues in persistent storage
- Processing of messages by message server(s)
  - May do filtering, transforming, logging, …
  - Networks of message servers
- But pretty low-level ('packet level') interactions, and still just point-to-point messages with no typing...
- Examples: IBM MQSeries, Java Message Service (JMS)

22

Note: not covered in lecture

Publish-Subscribe

- Get more flexibility with publish-subscribe:
  - **Publishers** advertise and publish **events**
  - **Subscribers** register interest in **topics** (i.e. a set of properties of events)
  - **Event-service** notifies interested subscribers of published events
- Keeps asynchronous (decoupled) nature of message-oriented middleware but:
  - Allows 1-to-many communication
  - Dynamic membership (publishers and subscribers can join or leave at any time)

Note: not covered in lecture

Publish-Subscribe: Pros and Cons

- Pub/sub useful for 'ad hoc' systems such as embedded systems or sensor networks:
  - Client(s) can 'listen' for occasional events
  - Don't need to define semantics of entire system in advance (e.g. what to do if get event <X>)
- Leads to natural "reactive" programming:
  - when <X>, <Y> occur then do <Z>
  - event-driven systems like Apama can help understand business processes in real-time
- But:
  - Can be awkward to use if application doesn't fit
  - And difficult to make perform well…

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

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