Motivation

- In P2P systems data is added by many peers
  - Stays in the system as long as the peer is online
  - Many peers may actually add the same item (popular music, etc.)

- Important items should be replicated so that is does not disappear in case of a single peer failure (Redundancy)
  - Popular data should also be replicated to allow for faster access

- Copies of data should be synchronized and kept in their original form, as some of them might become corrupt (Preservation)
  - Transfer through noisy channels (especially via malicious peers)
Outline

- Application
  - Distributed backup solutions for central document collections
  - High availability of content in P2P networks
  - Load balancing

- Enabling Technologies
  - Erasure Codes
  - Byzantine Agreements

- Sample Systems
  - LOCKSS
  - OceanStore

Erasure Codes
Error-Correcting Codes

- Error correcting codes introduce redundancy to cope with
  - Transmission failures (e.g., packet loss)
  - Noisy channels
  - Storage failures (e.g., hardware breakdown, churn)

- Basic idea:
  - Encode information of length n in (n+k) symbols
  - The information can be recovered from any n of the (n+k) symbols

- Examples
  - Check sums detect and correct errors in noisy channels
  - RAID-5 storage systems (Parity bits)

---

Err-Mail

- Err-mail works just like e-mail, except
  - About half of all the mail gets lost.
  - Messages longer than 5 characters are illegal.
  - It is very expensive (similar to air-mail).

- Alice wants to send her telephone number (555629) to Bob

- Naïve approach
  - Split into two packets (555, 629) and send separately
  - Chances are, one of them gets lost
  - Even repetitive sending doesn't help much, Bob will receive redundant packets
  - Acknowledge messages by Bob are an option, but expensive
Alice devises the following scheme.

- She breaks her telephone number up into two parts \( a = 555 \), \( b = 629 \), and sends 2 messages — "A=555" and "B=629" — to Bob.
- She constructs a linear function, \( f(n) = a + (b - a)(n - 1) \), in this case \( f(n) = 555 + 74(n - 1) \).
- She computes the values \( f(3) \), \( f(4) \), and \( f(5) \), and then transmits three redundant messages: "C=703", "D=777" and "E=851".

Bob knows that the form of \( f(n) \) is \( f(n) = a + (b - a)(n - 1) \), where \( a \) and \( b \) are the two parts of the telephone number.

Now suppose Bob receives "D=777" and "E=851"

Bob can reconstruct Alice's phone number by computing the values of \( a \) and \( b \) from the values (\( f(4) \) and \( f(5) \)).

Bob can perform this procedure using any two err-mails, so the erasure code in this example has a rate of 40%.
Tornado Codes

- Important class of erasure codes for practical applications

- Characteristics
  - Easy coding/decoding: linear codes with explicit construction
  - Fast coding/decoding: each check bit depends on only a few message bits

  - J. W. Byers, M. Luby, M. Mitzenmacher: Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads. INFOCOM 1999

Forward Error Correction

- Scenario
  - Application sends a real-time data stream of symbols
  - Network experiences unpredictable losses of at most a fraction of $p$ symbols
  - We know the positions of the lost bits (packet indexes)

- Insurance policy
  - Let $n$ be the block length
  - Instead of sending $n$ symbols, place $(1-p)n$ symbols in each block
  - Fill block to length $n$ with $pn$ redundant symbols

- Scheme provides optimal loss protection if message symbols can be recovered from any set of $(1-p)n$ symbols in the block
Forward Error Correction

- Interleave message bits and check bits in a stream

Properties of a good code

- There should be “few” check bits

- Linear time encoding
  - Average degree on the left should be a small constant

- Easy error detection/decoding
  - Each set of message bits should influence many check bits
  - Existence of unshared neighbors
Tornado Codes - Basic Idea

- Tornado code model: bipartite graph
- Each message bit is used in only a few check bits
  - Low degree bipartite graph
  - Check bits are computed as orthogonal combination of message bits (usually XOR)

\[
c_6 = m_3 \oplus m_7
\]

Graph Theory: Expander Graphs

- Properties
  - Expansion: every small subset \((k \leq \alpha n)\) on left has many \((\geq \beta k)\) neighbors on right
  - Low degree – not technically part of the definition, but typically assumed
Expander Graphs: Construction

- Important parameters: size \((n)\), degree \((d)\), expansion \((b)\)

- Randomized constructions
  - A random \(d\)-regular graph is an expander with a high probability
  - Construct by choosing \(d\) random perfect matchings
    - Perfect matching: all nodes on the left side get exactly one edge to a node on the right side
    - Repeat \(d\) times: every node on the left side has \(d\) edges to the right side
  - Time consuming and cannot be stored compactly

- Explicit constructions
  - Cayley graphs, Ramanujan graphs etc
  - Typical technique – start with a small expander, apply operations to increase its size

Tornado codes

- Will use \(d\)-regular bipartite graphs with \((1-p)n\) nodes on the left and \(pn\) on the right (e.g., \(p = 0.5\))
- Will need \(b > d/2\) expansion.
Tornado codes: Encoding

- Why is it linear time?

Complements the sum modulo 2 of its neighbors

Tornado codes: Decoding

- Assume that all the check bits are intact
- Find a check bit such that only one of its neighbors is erased (an unshared neighbor)
- Fix the erased code, and repeat.
Tornado codes: Decoding

- Need to ensure that we can always find a check bit
- “Unshared neighbors” property
  - Consider the set of corrupted message bit and their neighbors.
  - Suppose this set is small \( \Rightarrow \) at least one message bit has an unshared neighbor.
- Can we always find unshared neighbors?
  - Theorem: Expander graphs give us this property if \( b > d/2 \)

What if check bits are lost?

- Cascading
  - Use another bipartite graph to construct another level of check bits for the check bits
  - Final level is encoded by some other code, e.g., Reed-Solomon
Byzantine Agreements

Byzantine Generals Problem

- n generals are planning a coordinated attack against a common enemy
  - generals located in different places
  - each general has initial opinion on whether ready to attack
  - some generals may be treacherous
  - if all "good" generals attack ➔ success, otherwise disaster

- Is it possible for the good generals to agree on whether to attack or not, without knowing a priori who the treacherous generals are?
  - If so, a protocol for reaching an agreement can be designed
Correctness Conditions for Byzantine Consensus

- **Agreement**
  - No two “good” generals agree on different outcomes

- **Validity**
  - If all “good” generals start out with the belief they are ready to attack, then the only possible outcome is to attack

- **Termination**
  - All “good” generals eventually decide

- Generals could be peers, database nodes, circuit switches, etc.

---

Upper Limit on Number of Bad Generals?

- For what percentage of malicious peers can protocols be designed?

- **Triple Modular Redundancy \( \geq 3f \) nodes**
  - Assuming \( f \) treacherous generals (malicious peers), we need at least \((3f+1)\) peers to come to an agreement

Example: Only Three Peers with One Traitor

- Consider system with 3 peers
  - Each starts with an initial value (0 or 1)
  - One peer is malicious
  - Good nodes need to agree upon value (0 or 1)
- Nodes act solely based on messages coming in along incident edges
- Assume there exists an algorithm that allows good nodes to agree

![Diagram of three peers](image)

Example: Only Three Peers with One Traitor

- Assume that P₁ is a good peer
- **Scenario 1: P₃ is treacherous**
  - P₂ relates that it is in state 0 to P₁ and P₃
  - But P₃ relates to P₁ that P₂ is in state 1
- **Scenario 2: P₂ is treacherous**
  - P₂ relates that it is in state 0 to P₁ and that it is in state 1 to P₃
  - P₃ relates to P₁ that P₂ is in state 1
- Obviously P₁ cannot distinguish the two scenarios
  - In both cases it would have to decide for a value of 0 for the respective loyal peer
Example: Only Three Peers with One Traitor

- Now look at P₃ in scenario 2
- Remember in scenario 2 *P₂ is treacherous*
  - P₂ relates that it is in state 0 to P₁ and that it is in state 1 to P₃
  - P₁ relates to P₃ that it is in state 1
- P₃ would have to decide for 1 to and thus vote with the loyal peer P₁

- **Contradiction:** in scenario 2 P₁ and P₃ would both be loyal, but would still vote differently

Solution for the Byzantine Agreement (n > 3f)

- One peer starts the agreement process by broadcasting its value (commander)
  - Whenever a message is supposed to be sent, but a peer does not send it, it is detected, and a default value is assumed
- Echo the result to all other peers
- Do this for more peers than can be malicious
  - Algorithm is recursive with (f+1) levels

- **Bottom case:** no traitors
  - the commander broadcasts its initial value
  - every other process decides on the value it receives
Solution for the Byzantine Agreement

- **Idea:**
  Amplify the original message over different channels starting from \( (f+1) \) commanders

- **Solution for the Byzantine Agreement \((n > 3f)\)**

  - \texttt{echo\_broadcast(peer C, message m)}
    - \(C\) sends \([\text{initial},C,m]\) to all peers
    - Every recipient replies with \([\text{echo},C,m]\) to all and ignores subsequent \([\text{initial},C,m']\)
    - Upon receiving \([\text{echo},C,m]\) from \((n+f)/2\) distinct peers, then a peer accepts \(m\) from \(C\)

  - **Terminates? Yes** — all non-malicious peers accept \((n-f)\) messages and exit both \textit{wait} phases.
  - If the system is initially proper (all non-malicious peers have the same value \(m\)) then every such peer terminates the algorithm with \(M=m\).
Solution for the Byzantine Agreement (n > 3f)

\[ C_i \cdot M_i := M_j \]

for \( k = 1 \) to \((f+1)\) do

(* Phase 1: SEND *)
- broadcast \( M \);
- wait to receive \( M \)-messages from \((n-f)\) distinct processes;
- \( proof := \text{set of received messages}; \)
- \( count(1) := \text{number of received messages with } M = 1; \)
- \( \text{if } count(1) > (n-2f) \text{ then } M := 1; \)
- \( \text{else } M := 0; \)

(* Phase 2: ECHO *)
- echo_broadcast \([M, proof]\);
- wait to accept \([M, proof]\)-messages, with a correct proof, from \((n-f)\) distinct processes;
- \( count(1) := \text{number of accepted messages with } M = 1; \)
- Compute_new_vote( \( s_k \));
- \( \text{if } (s_k = 0 \text{ and } count(1) \geq 1) \text{ or } (s_k = 1 \text{ and } count(1) \geq (2f+1)) \text{ then } M := 1; \)
- \( \text{else } M := 0; \)

Example: Four Generals

- If the Commander is not malicious (agreement by majority vote)

```
\[ C \]

\[ \text{V} \quad \text{V} \quad \text{V} \quad \text{V} \quad \text{V} \quad \text{V} \quad \text{V} \quad \text{V} \quad \text{?} \quad \text{?} \]
```

L3S Research Center P2P Content Distribution 31

L3S Research Center P2P Content Distribution 32
Example: Four Generals

- If the Commander is malicious (no agreement possible)

![Diagram of four generals example]

Generalization for Byzantine Agreement

- Partition peers into three groups, with at least 1 and at most 1/3 of the peers in each group
- Theorem: A Byzantine agreement can be solved in a network $G$ of $n$ peers while tolerating $f$ faults if and only if
  - $n > 3f$ and
  - $\text{connectivity}(G) > 2f$
- Graph $G$ is $2f$-connected if the removal of $2f$ or more peers will result in a disconnected graph (or a trivial 1-node graph)
LOCKSS (HP Labs)

Stands for: Lots of Copies Keep Stuff Safe
- Basic idea is that distributing copies over the network will make access easy and keep material online, even in face of peer faults
- www.lockss.org

LOCKSS is not an archive
- Archives are for materials that are hard to replicate
- Sacrifice access to ensure preservation

But rather a global library system
- Libraries for easily replicated materials
- Ensure access at some preservation risk

Central Question: How do you ensure that copies in the system are not compromised?
LOCKSS Assumptions

- Be affordable
  - Cheap PC, open-source software
  - Low administration “appliance”
- Have low probability of failure
  - Many replicas, resists attack, no secrets
  - Scale to enormous rates of publishing
- Preserve access
  - Links resolve, searches work
  - Conform to publishers access controls
- Libraries take custody of content

Why is Long-Term Storage Hard?

- Large-scale disaster
- Human error
- Media faults
- Component faults
- Economic faults
- Attack
- Organizational faults
- Media/hardware obsolescence
- Software/format obsolescence
- Lost context/metadata
Strategies for Dealing with this Mess

- Address high costs of preservation
  - Commodity hardware
  - Reduce on-going costs
  - Better cost models
- Replicate content, break correlations between replicas
  - Geographic, administrative, platform, media, formats…
- Audit replicas proactively to detect damage
  - Data must be accessible to do this cheaply!
- Migrate content to maintain usability
  - To new hardware, formats, keys…
- Avoid external dependencies
  - Includes vendor lock-in, DRM issues
- Plan for data exit

Exploit Existing Replication

- Testbed: electronic journals in libraries
- Many libraries subscribe to the same materials
- Appliances used by libraries around the world
  - Cheap PC with some storage
  - Libraries maintain existing relationships with publishers
  - Materials are subscribed to be collected/preserved
  - Run a P2P audit/repair protocol between LOCKSS peers
  - Not a file sharing application
- Survive or degrade gracefully in the face of
  - Latent storage faults & sustained attacks
- Make it hard to change consensus of population
How does it Actually Work?

- The LOCKSS audit/repair protocol

- A peer periodically audits its own content
  - To check its integrity
  - Calls an opinion poll on its content every 3 months
  - Gathers repairs from peers

- Raises alarm when it suspects an attack
  - Correlated failures
  - IP address spoofing
  - System slowdown

Sampled Opinion Poll

- Each peer holds for each document
  - Reference list of peers it has discovered
  - History of interactions with others (balance of contributions)

- Periodically (faster than rate of storage failures)
  - Poller takes a random sample of the peers in its reference list
  - Invites them to vote: send a hash of their replica

- Compares votes with its local copy
  - Overwhelming agreement (>70%) → Sleep blissfully
  - Overwhelming disagreement (<30%) → Repair
  - Too close to call → Raise an alarm

- Repair: peer gets pieces of replica from disagreeing peers
  - Re-evaluates the same votes

- Every peer is both poller and voter
Bimodal Alarm Behavior

- Most replicas the same
  - No alarms
- In between
  - Alarms very likely
- To achieve corruption
  - Adversary must pass
- through “moat” of alarming states
  - Damaged peers vote
- with undamaged peers
  - Rate limitation helps

Probability of Irrecoverable Damage

Preservation succeeds for up to 35% subversion
  - For powerful attacker (unlimited CPU/identities)
  - Attacking for 30 years
OceanStore (UC Berkeley)

Real-World Computing Applications

- Peer-to-peer networks
  - Improve availability through wide replication
  - Untrusted decentralized infrastructure
- Oceanstore: provide long-time available data
  - Inner ring holds committed data uses byzantine agreement
  - Target is global scale data access
  - http://oceanstore.cs.berkeley.edu/
Ubiquitous Devices → Ubiquitous Storage

- Consumers of data move, change from one device to another, work in cafes, cars, airplanes, the office, etc.
- Properties required for OceanStore storage
  - **Strong Security**: data encrypted in the infrastructure; resistance to monitoring and denial of service attacks
  - **Coherence**: too much data for naive users to keep coherent “by hand”
  - **Automatic replica management and optimization**: huge quantities of data cannot be managed manually
  - **Simple and automatic recovery from disasters**: probability of failure increases with size of system
  - **Utility model**: world-scale system requires cooperation across administrative boundaries

OceanStore: Everyone’s Data, One Big Utility

- “The data is just out there”
- **Separate information from location**
  - Locality is an only an optimization
  - Wide-scale coding and replication for durability
- **All information is globally identified**
  - Unique identifiers are hashes over names & keys
  - Single uniform lookup interface replaces: DNS, server location, data location
  - No centralized namespace required
**OceanStore Assumptions**

- **Untrusted Infrastructure:**
  - The OceanStore is comprised of untrusted components
  - Only cyphertext within the infrastructure
  - Information must not be “leaked” over time

- **Mostly Well-Connected:**
  - Data producers and consumers are connected to a high-bandwidth network most of the time
  - Exploit multicast for quicker consistency when possible

- **Promiscuous Caching:**
  - Data may be cached anywhere, anytime

- **Trusted party is responsible for keeping up service**

---

**Questions about Information**

- **Where is persistent information stored?**
  - **Wanted:** Geographic independence for availability, durability, and freedom to adapt to circumstances

- **How is it protected?**
  - **Wanted:** Encryption for privacy, signatures for authenticity, and Byzantine commitment for integrity

- **Can we make it indestructible?**
  - **Wanted:** Redundancy with continuous repair and redistribution for long-term durability

- **Is it hard to manage?**
  - **Wanted:** Automatic optimization, diagnosis and repair
Naming and Data Location

- **Requirements:**
  - System-level names should help to authenticate data
  - Route to nearby data without global communication
  - Don’t inhibit rapid relocation of data
- **Approach:** Two-level search with embedded routing
  - Underlying namespace is flat and built from secure cryptographic hashes (160-bit SHA-1)
  - Search process combines quick, probabilistic search with slower guaranteed search
  - Long-distance data location and routing are integrated
    - Every source/destination pair has multiple routing paths
    - Continuous, on-line optimization adapts for hot spots, denial of service, and inefficiencies in routing

Rapid Update in an Untrusted Infrastructure

- **Requirements:**
  - Scalable coherence mechanism which can operate directly on encrypted data without revealing information
  - Handle Byzantine failures
  - Rapid dissemination of committed information
- **OceanStore Approach:**
  - Operations-based interface using conflict resolution
    - Modeled after Xerox Bayou ⇒ updates packets include:
      - Predicate/update pairs which operate on encrypted data
    - Use of oblivious function techniques to perform this update
    - Use of incremental cryptographic techniques
  - User signs Updates and trusted party signs commits
  - Committed data multicast to clients
Tentative Updates: Epidemic Dissemination

Committed Updates: Multicast Dissemination
Oceanstore: State of the Art

- Techniques for protecting metadata
  - Uses encryption and signatures to provide protection against substitution attacks
- Working scheme that can do some forms of conflict resolution directly on encrypted data
  - Uses new technique for searching on encrypted data.
  - Can be generalized to perform optimistic concurrency, but at cost in performance and possibly privacy
- Byzantine assumptions for update commitment
  - Signatures on update requests from clients
    - Compromised servers are unable to produce valid updates
    - Uncompromised second-tier servers can make consistent ordering decision with respect to tentative commits

High-Availability and Disaster Recovery

- Requirements:
  - Handle diverse, unstable participants in OceanStore
  - Mitigate denial of service attacks
  - Eliminate backup as independent (and fallible) technology
  - Flexible “disaster recovery” for everyone
- OceanStore Approach:
  - Use of erasure-codes to provide stable storage for archival copies and snapshots of live data
  - Version-based update for painless recovery
  - Continuous introspection repairs data structures and degree of redundancy
**Archival Dissemination of Fragments**

![Image of a network diagram]

**Automatic Maintenance**

- **Byzantine Commitment for inner ring:**
  - Can tolerate up to 1/3 faulty servers in inner ring
    - Bad servers can be arbitrarily bad
    - Cost $\sim n^2$ communication
  - Continuous refresh of set of inner-ring servers
    - Proactive threshold signatures
    - Use of Tapestry $\Rightarrow$ membership of inner ring unknown to clients

- **Secondary tier self-organized into overlay dissemination tree**
  - Use of Tapestry routing to suggest placement of replicas in the infrastructure
  - Automatic choice between update vs invalidate