

Peer-to-Peer Systems

Security and Reliability

Michael Welzl michael.welzl@uibk.ac.at

DPS NSG Team <http://dps.uibk.ac.at/nsg>
Institute of Computer Science
University of Innsbruck, Austria

Outline

- This block addresses the question:
[What if something goes wrong?](#)
- What can go wrong?
 - Attackers can infiltrate the system
 - Nodes can fail
- Why would nodes fail?
 - Technical reasons (e.g. link outage)
 - Denial-of-Service attacks
 - Censorship
- Reliability / resilience and security are related issues

Security in DHTs

- DHT architectures assumes a trusted system
 - True in corporate environments, but not on the Internet
- One solution: Central certificate-granting authority
 - Used by Pastry and its related projects
 - Constrains membership in DHT
- One attack: Return incorrect data
 - Easy to avoid through cryptographic techniques
 - Detect and ignore non-authentic data
- Focus: Attacks that prevent participants from finding the data
 - Threatens the liveliness of the system

DHT Components and adversary model

DHTs have the following components:

1. Key identifier space
 2. Node identifier space
 3. Rules for associating keys to nodes
 4. Per-node routing tables that refer to other nodes
 5. Rules for updating routing tables as nodes join and leave
- Any of the above may be the target of the attack from an adversary
 - Adversaries are participants in DHT that do not follow protocol correctly

Adversary model - assumptions:

- Malicious node can generate arbitrary packets
 - Includes forged source IP address
- Can receive only packets addressed to itself
 - Not able to overhear communications between other nodes
- Malicious nodes can conspire together, but still limited as above

Types of Attacks

1. Routing attacks
 2. Attack against data storage
 3. Miscellaneous attacks
- First goal: **Detect attack**
 - Violation of invariants or contracts
 - What to do when an attack is detected?
 - Is other node malicious?
 - Did other node simply not detect attack?
 - Achieving **verifiability** is vital

Routing Attacks

- Routing is responsible for maintaining routing tables and sending messages to correct nodes
 - Routing **must** function correctly
 - **Define invariants and check them**
- **Attacker can incorrectly forward messages**
 - But: Each hop should get "closer" to destination
 - Querying node should check this
 - **Allow querying node to observe lookup process**
 - For example, processing messages recursively hides this
- **Attacker can claim that wrong node is responsible node**
 - Querying node is "far away", cannot verify this
 - **Assign keys to nodes in a verifiable way**
 - Often: Assign node IDs in a verifiable way (e.g., IP address)
 - For example, CAN lets node pick its own ID...

More Routing Attacks

- Attacker can send incorrect routing updates
 - Blatantly wrong updates can be detected
 - If DHT allows several choices for next hop
 - Attacker can pick a "bad" node
 - Not necessarily a problem with correctness, only performance
 - Can be a problem for some applications (anonymity)
 - Server selection can be abused
- Attacker can partition network
 - If new node contacts attacker first, attacker can partition network (can even hijack nodes from real network)
 - Parallel network is consistent and "looks OK"
 - Attacker can track nodes
 - Bootstrap from a trusted source: Hard to get in dynamic networks, public keys might help
 - Cross check routing tables with random queries
 - Assumes we were part of network earlier, still not totally safe

Storage and Retrieval Attacks

- Attacker can deny existence of data
 - Or return wrong data
- Must implement replication at storage layer
 - Who creates replicas?
 - Clients must be able to verify that all copies were created
- Avoid single points of responsibility
 - Replication with multiple hash functions is one good way
- Big problem if system does not verify IDs
 - Any node can become responsible for any data
 - For example, Chord allows virtual nodes

Miscellaneous Attacks

- Attacker can behave inconsistently
 - Some nodes see it as good, others as bad
 - Maintain good face to nearby nodes
 - How would a distant node convince neighbors of bad node?
 - Public keys and signatures could solve this
- Denial of service
 - Attacker floods a node with messages
 - Node appears failed to the rest of the network
 - Replication helps, but attacker may succeed if replication not sufficient
 - Replicas should be in physically different locations
 - DHT assigns keys to nodes randomly, should be OK
 - Large attacks require lot of resources

More Miscellaneous Attacks

- Attacker can join and leave the network rapidly
 - Causes lot of stabilization traffic in network
 - Loss of performance, maybe loss of correctness
 - Works well if stabilization requires lot of data transfer
 - For example, copying of large objects from node to node
 - DHT must handle this case anyway
- Attacker can send unsolicited messages
 - Q asks E and gets referred to A
 - E knows Q expects an answer from A
 - E forges message from A to Q
 - Public keys and signatures (heavy solution)
 - Random nonce in a message works also

Design Principles

Summary of design principles for secure DHT:

1. Define verifiable system invariants (and verify them!)
2. Allow querying node to observe lookup process
3. Assign keys to nodes in a verifiable way
4. Server selection in routing may be abused
5. Cross-check routing tables with random queries
6. Avoid single points of responsibility

Sybil Attack

- **Entity:** Real-world entity; **Identity:** Representation in the system
- Redundancy requires resources to be spread across several entities
 - Peer-to-peer systems work only with identities
- **Sybil Attack:** one entity creates multiple identities to attack the system
 - From book/movie telling the story of Sybil Isabel Dorsett who suffered from multiple personality disorder
- For example, data replication
 - A single copy might be on a malicious peer
 - But several copies on different peers are safe, right?
- How can we know that the “different” peers are really different and distinct physical entities?
- **Answer:** We need a (logically) centralized, trusted entity (e.g., CA)
 - Without central authority, problem was proven to be *unsolvable*

Examples of Solutions

- Real centralized authorities:
 - Certification Authorities, e.g., VeriSign
- Logically centralized authorities:
 - Hashing IP address to get DHT identifier (e.g., CFS)
 - Add host identifiers to DNS names (SFS)
 - Cryptographic keys in hardware (EMBASSY)
 - These appear distributed, but they all rely on some centralized authority (e.g., ICANN gives out IP addresses and DNS names)
- Identities vouching for other identities
 - For example, PGP web of trust for humans
 - **NOT a solution!**
 - Attacker can attack the system early and compromise generation of identities and break chain of vouchers

Results

- Entity should accept identities only if they have been validated by central authority, itself, or others
 - In a fully distributed system, only entity itself and others
- The following can be shown under reasonably realistic assumptions for direct validation:
 1. Even when severely resource constrained, a faulty entity can counterfeit a constant number of multiple identities
 2. Each correct entity must simultaneously validate all the identities it is presented; otherwise, a faulty entity can counterfeit an unbounded number of entities
 - Similar results hold for indirect validation by others
- What resources can be used in identification?
 - Communication, CPU, storage

Resources as Proof

- **Communication**
 - Broadcast request for others to identify themselves and accept only responses which come within a certain time interval
 - Model had assumed broadcast communications
- **CPU**
 - Require other peer to perform some computationally intensive, but easily verifiable, task
 - This requires simultaneous identification (point 2 from above)
- **Storage**
 - Have others store some uncompressible data and periodically ask them to give back a small piece
 - Would eventually catch a Sybil attack
 - Problem: No storage space left for doing any real work...

Implications of Sybil Attack

- Need centralized authority for managing identities
- Logically centralized systems should be aware of their potential (future) vulnerabilities
 - For example, privacy extensions for IPv6 might break CFS
- Sybil attack can be avoided under the assumptions:
 - All entities operate under identical resource constraints
 - All presented identities are validated simultaneously by all entities, coordinated over the whole system
 - For indirect validation, the number of vouchers must exceed the number of failures in system
- Are these assumptions feasible or practical for a large-scale distributed system?
 - Answer would seem to be no

Byzantine Generals Problem

- Several divisions of the Byzantine army surround an enemy city. Each division is commanded by a general.
- The generals communicate only through messenger
 - Need to arrive at a common plan after observing the enemy
- Some of the generals may be traitors
 - Traitors can send false messages
- Required: An algorithm to guarantee that
 1. All loyal generals decide upon the same plan of action, irrespective of what the traitors do.
 2. A small number of traitors cannot cause the loyal generals to adopt a bad plan.

Theorem: (from L. Lamport)

- If no more than m generals out of $n = 3m + 1$ are traitors, everybody will follow the orders

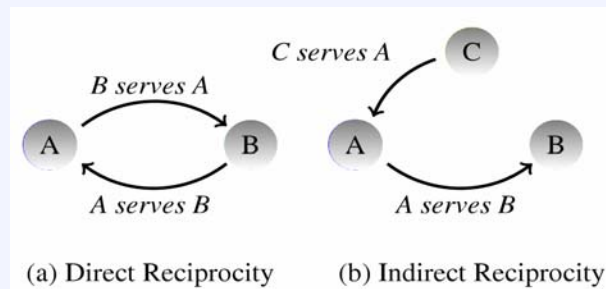
Solution:

- Digital signatures; all generals sign their orders, inconsistent orders or forwarding of false messages can immediately be detected

Soft-security solutions

- What is the common goal?
 - Nodes must fulfill their obligation toward the system and other nodes
 - Incentive cooperation
- Theoretical approaches:
 - Mechanism design
 - Game theory
- Simplifications must be made to study the complexity of networked systems
 - Useful to understand the behaviour of rational nodes
- Monetary scheme
 - Needs to have tamper-proof hardware
 - Accounting infrastructure
- Service Differentiation

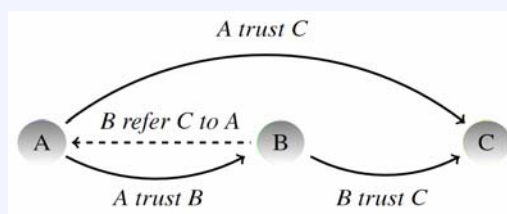
Social science



- Reciprocal altruism: entities do not expect any service in return
- Indirect reciprocity only possible if transactions are monitored

Reputation

- Peer-to-peer systems must create and maintain trust to function properly
 - Trust is users' knowledge about the reliability of authenticated parties
- Reputation is an important component of all human (and machine) interactions



Trust transitivity

Reputation Management Systems

- Create a framework to foster cooperation
- Provide a sense of trust to nodes that are willing to cooperate
- Reputation management systems require three properties:
 - Nodes should last for long in the system
 - Nodes should distribute feedback
 - Feedbacks should be useful to the community
- Additional desirable properties
 - Anonymity
 - Minimal overhead (storage, computation, messages)

Reputation Management Systems /2

- Various aspects to be considered regarding trust information
 - how to aggregate
 - where to store
 - how to collect (proactive vs. reactive)
- e.g. **ROCQ** scheme defines **reputation** and **credibility** of a node
 - as well as its **opinion** and the **quality** of this opinion (confidence level)
- Other issues of reputation management systems
 - communication overhead
 - Content Distribution Networks (CDNs): other credibility considerations needed for parts of files (whole file becomes useless if a part is a fake)
 - mobility: requirement of long lasting nodes broken as nodes move around
 - can be solved by letting a node carry its own reputation as a verifiable token

Reliability / performance issues with DHTs

- DHTs trade-offs: performance vs. cost, reliability vs. cost
 - Cost: node state or number of bytes sent into network
 - Reliability / performance often connected
 - Performance = lookup latency = $f(\text{number of hops})$
 - Assume random failures: long path = unreliable system
- It was shown that we can configure a DHT to give us “decent” performance (lookup latency) at “reasonable” cost (overlay maintenance overhead)
- Question: Is “decent” good enough for real applications?
- In other words, how does a DHT-based P2P application compare against a client/server-application?
- Let’s take Domain Name System (DNS) as example
 - Fundamental Internet-service
 - Very much a client/server application

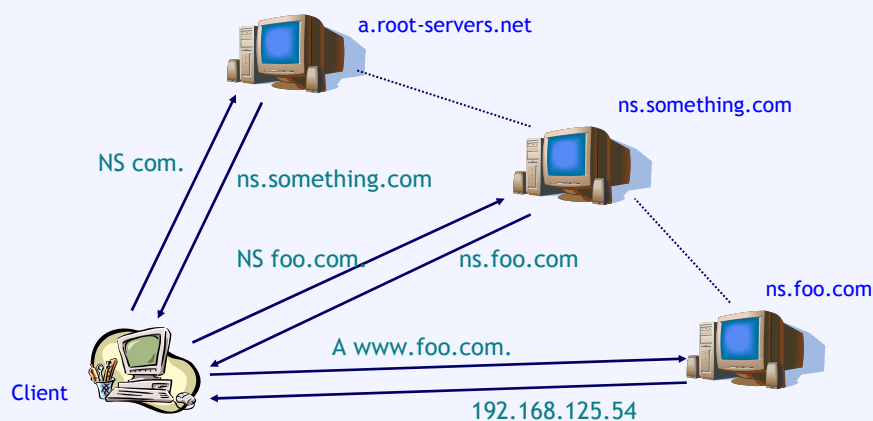
P2P DNS

- Domain Name System (DNS) very much client-server
- *Ownership of domain = responsibility to serve its data*
- DNS concentrates traffic on root servers
 - Up to 18% of DNS traffic goes to root servers
- A lot of traffic also due to misconfigurations
- P2P DNS
 - puts expertise in the system
 - No need to be an expert administrator
 - shares load more equally
- So why not replace standard DNS with P2P DNS?

DNS: Overview

- DNS organized in zones (\approx domain)
 - Actual data in resource records (RR)
 - Several types of RRs: A, PTR, NS, MX, CNAME, ...
- Administrator of zone responsible for setting up a server for that zone (+ redundant servers at other domains)
- Queries resolved hierarchically, starting from root
- Owner of a zone is responsible for serving zone's data
- DNS shortcomings:
 - Need skill to configure name server
 - No security (but added later to some degree)
 - Queries can take very long in worst case

DNS: Example



- Client wants to resolve `www.foo.com`
- Replies to queries have additional information (IP address + name)
- Queries can be iterative (here) or recursive

How to Do P2P DNS?

- Put DNS resource records in a DHT
- Key is hash of domain name and query type
 - For example, SHA1(www.foo.com, A)
- Values replicated for better performance (~ 5-7 copies)
- Can be built on any DHT, works the same way
- All resource records must be signed
 - Some overhead for key retrieval
- For migration, put P2P DNS server on local machine
 - Configure normal DNS to go through P2P DNS
 - No difference to applications

P2P DNS: Performance

- Current DNS has median latency of 43 ms
 - Measured at MIT
- Some queries can take a long time
 - Up to 1 minute (due to default timeouts)
- P2P DNS has median latency of 350 ms!
 - Simulated on top of Chord
- Conclusion:
P2P DNS is much, much worse than standard DNS
 - But extremely long queries cannot happen

Why (not) P2P DNS?

Pros

- Simpler administration
 - Most problems in current DNS are misconfigurations
 - DNS servers not easy to configure well
- P2P DNS robust against lost network connectivity
 - Current DNS: first DNS server unavailable \Rightarrow all lookups fail
- No risk of incorrect delegation
 - Subdomains can be easily established
 - Signatures confirm

Cons

- All queries must be anticipated in advance
 - With current DNS, a local database could be queried as a request arrives
- Current DNS can tailor requests to client
 - Widely used in content distribution networks and load balancing
- Might be possible to implement above in client software
- But latency problem remains!

Future of DHT-Based Applications?

- DHT-based applications have to make several RPCs
 - 1 million node Chord = 20 RPCs, Tapestry 5 RPCs
- Experiments with DNS show even 5 is too much
 - Current DNS usually needs 2 RPCs
 - DNS puts a lot of knowledge at the top of the hierarchy
 - Root servers know about millions of domains
- Many RPCs is main weakness of DHTs
- DHT-based applications have all their features on clients
 - New feature \Rightarrow install new clients
 - Some kind of an "active" network as a solution?

Reliability of P2P Storage

- Example case: P2P storage system
 - Each object replicated in some peers
 - Peers can find where objects should be
 - Typically DHT-based, but DHT is not absolutely required
- No concern of consistency
 - Read-only storage system

Questions:

1. How many copies are needed for a given level of reliability?
 - Unconstrained system with infinite resources
2. What is the optimal number of copies?
 - System with storage constraints

Reliability of Data in DHT-Storage

- Storage system using a distributed hash table (DHT)
- Peer A wants to store object O
 - Create k copies on different peers
 - k peers determined by DHT for each object (k closest)
- Later peer B wants to read O
 - What can go wrong?
- Simple storage system: Object created once, read many times, no modifications to object
- Question: What is the value of k needed to achieve e.g., 99.9% availability of O ?
 - Remember: Only probabilistic guarantees possible!

Assumptions

- Assume I peers in the DHT
 - Each peer has unlimited storage capacity
- Peer is up with probability p
 - Peers are homogeneous, i.e., all peers have same up-probability
- Peers uniformly distributed in hash space
 - Makes mathematical analysis tractable
- New peers can join the network
- Peers never permanently leave
- User may need to access several objects to complete one user-level action
 - For example, resolve path to file

What Can Go Wrong?

1. All k peers are down when B reads
 - Object is not available in any on-line peer
 2. k closest peers were down when A wrote and are up when B reads
 3. At least k peers join and become new closest peers
 - In above two cases, object is (maybe) still available in the peers where A wrote it
 4. All k peers have permanently left the network
 - Assumed not to happen
- We only look at the first three cases
 - What are the probabilities of each one of them?

Probabilities of Loss

1. All k peers are down when B reads

$$p_{l1} = (1 - p)^k$$

2. k closest peers were down when A wrote and are up when B reads

$$p_{l2} \approx \sum_{i=k}^{(1-p)I} \binom{(1-p)I}{i} \left(\frac{p(1-p)}{I} \right)^i$$

3. N peers join and at least k peers become new closest peers

$$p_{l3} = \sum_{i=k}^N \binom{N}{i} \frac{1}{I^i}$$

Numerical Values for Loss

		I		
		10^2	10^3	10^4
$p_{l3} \approx$		10^{-10}	10^{-15}	10^{-20}

p	$p_{l1} \approx$	$p_{l2} \approx (\text{for given } I \text{ and } p)$		
0.99	10^{-10}	0	10^{-15}	10^{-15}
0.9	10^{-8}	10^{-8}	10^{-8}	10^{-8}
0.5	0.03	10^{-4}	10^{-4}	10^{-4}
0.3	0.17	10^{-3}	10^{-3}	10^{-3}

- First case clearly dominates
 - In above tables, $k = 5$
 - Cases 2 and 3 may look good at first sight, but note:
Often necessary to search more than k nodes to find object!

How to Improve?

- Maintain storage invariant $\rightarrow O$ always at k closest
 - Needs additional coordination
 - Possible if down-events controlled
 - Crash \rightarrow others need to detect crash (before they crash)
 - Guarantees availability as long as invariant maintained
 - Possibly wastes storage if copies are not removed when peers come back into the system
 - This approach taken by PAST storage system
- Increase k
 - Create more copies, simple to implement
 - Wastes storage capacity?
 - Not good for changing objects (consistency)

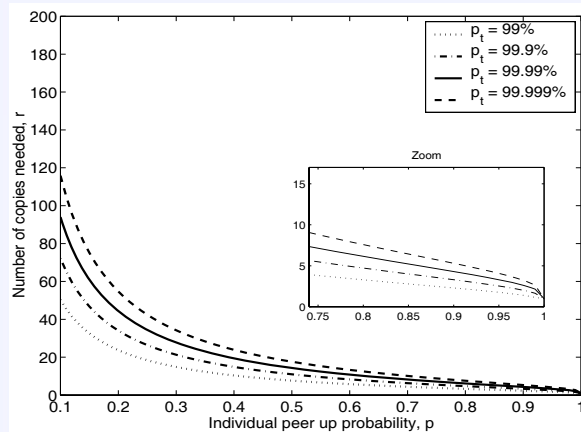
What does the user see?

- Suppose: User's action needs to access several objects
 - For example, resolve path for files one level at a time
- For each object: $p_s = 1 - p_{\text{fail}} = 1 - (1 - p)^k$
- What if we need to access 2 objects?
- Success for user: $p_t = (1 - (1 - p)^k)^2$
- Solving for k :

$$k = \frac{\log(1 - \sqrt{p_t})}{\log(1 - p)}$$

- In general for n objects: $p_t = (1 - (1 - p)^k)^n$

How Large Should k Be?



- Define target p_t
 - This is what user sees
 - Failures **temporary**
- When peers mostly up, k small
- Increase in $p_t \rightarrow$ small increase in k

Replication: how to do it?

- Replication in read-only system helps availability
- Main cause of unavailability is k peers being down at the same time when trying to read
- Create k copies of each object
 - If peers mostly up, k quite small (< 10)
 - Actively maintaining copies in right peers helps
- Where to place objects?
- Key assumption of DHTs: load evenly distributed across address space
 - Then storing replicas in local neighbors will preserve this property

Two replication strategies for Chord

1. Successor list

- Chord maintains 1 successor pointer, 1 predecessor pointer, finger table
- Idea for storing replicas in (overlay) proximity:
 - Maintain pointers to next S successors
($N*(S-1)$) additional pointers in the whole system)
 - Store replica in all these nodes
 - Maintenance: copy / move replica as nodes come and go (or fail)

2. Multiple nodes in one interval

- Assign interval responsibility to more than one node
- Each node stores additional pointers to neighbors in the same interval
 - But only one finger pointer
- Joining node announces itself to nodes responsible for the same interval

Shortcomings of proximity based methods

- Proximity based replica storage assumes that objects evenly distributed across address space
 - Not always true
- Prior analysis assumes all objects equally popular or important
 - Not always true
 - Zipf-distribution for object popularities
 - Also, some objects may require higher availability
- How should objects be replicated in this case?
- Algorithms based on notion of well connected P2P community (e.g. campus)
 - Replacement policies such as Most Frequently Requested (MFR)
 - Each object o has "attractor nodes"; Object o tends to get replicated in its attractor nodes; Queries for o tend to be sent to attractor nodes \Rightarrow tend to get hits

Redundancy

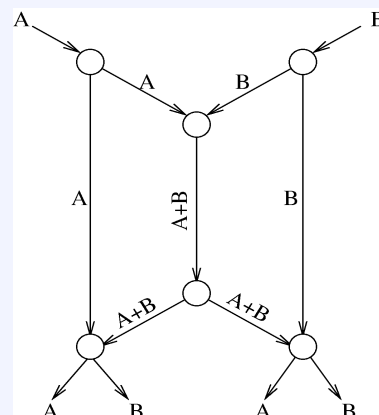
- No need to always replicate full objects
 - What if parts of objects are distributed?
 - We can go beyond simple splitting...
- Erasure codes
 - Split each object into N fragments
 - Compute K redundant fragments
 - Disseminate these $N+K$ blocks
 - Any N out of these $N+K$ blocks suffice for reconstructing the object
- Most efficient and common method: **network coding**
 - Based on linear combinations of orthogonal vectors in finite fields
 - But easier to explain with XOR :-)
 - Network coding applied for numerous things nowadays (e.g. mobile nets)

Network coding

R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network Information Flow", (*IEEE Transactions on Information Theory*, IT-46, pp. 1204-1216, 2000)

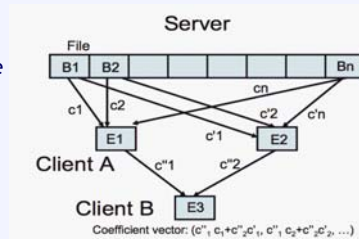
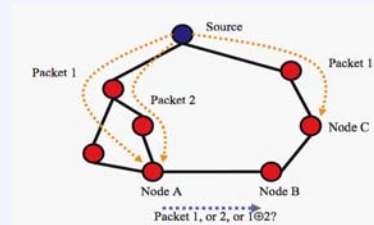
Example:

- Bits A and B should be transmitted
- Only one bit can be sent on each link
 - Simply send the bits: left side gets A , right side gets B , one of them can get the other in addition via middle link
 - By sending $A \text{ XOR } B$, both sides can get A and B in one step



Practical Network Coding

- Avalanche (Gkantsidis, Rodriguez, 2005)
- Goal
 - Avoid Coupon-Collector-Problem when getting blocks of an object
 - Calculation of how often to buy in order to get all 10 different coupons
 - Problem does not need to arise with network coding: an object consisting of m parts can be reconstructed from any m parts
 - This is closer to most coupons in real life...
 - Optimal dissemination of data regarding available bandwidth
- Method
 - Disseminate linear combinations of object parts
 - Receiver collects everything, then reconstructs original object



Pro's and con's of network coding

- Major performance and reliability gains claimed for a multitude of things
- But: significant overhead
- Storage overhead
 - E.g. 4 GB file with 100 KB block must contain variable vector of 4 GB/100 KB = 40 KB = 40% overhead per block
 - Better: 4 GByte and 1 MByte-Block; resulting overhead per block 4 KB = 0,4%
- Decoding: memory and CPU
 - Inverting a $m \times m$ -matrix (m = size of variable vector)
 - this needs time $O(m^3)$ and memory $O(m^2)$
- Read-/write-access to files
 - Encoding / decoding: for m blocks, must traverse whole file m times
 - Disk cache cannot be exploited because no data locality

Conclusion

- Security and reliability are major issues in P2P systems
 - They are related
- Reliability is also related to performance
 - Avoid long paths: more reliable, shorter lookup latency
 - Network coding: can improve reliability and performance
- A lot of unresolved issues and open questions
 - How to efficiently cope with Sybil attacks
 - Ongoing work on reputation management systems
 - How to ideally replicate (depending on distribution of popularity items)
 - Trade-off between redundancy and replication
 - Will network coding prevail?