Lecture #12:
Data Availability

COMS 4995-001:
The Science of Blockchains

URL: https://timroughgarden.org/s25/

Tim Roughgarden

Goals for Lecture #12

1. Data availability committees.
   – offload storage responsibilities from validators to third parties
2. Verifiable information dispersal (VID).
   – use Merkle trees and Reed-Solomon codes to get minimal overhead
3. Data availability sampling.
   – how can an end user be confident that data is available?
   – idea: repeatedly download small random chunks

Validator Responsibilities, Revisited

Consensus: agree on a sequence blocks/transactions.

Execution: process txs, keep state of blockchain up-to-date.

Validator Responsibilities, Revisited

Consensus: agree on a sequence blocks/transactions.

Execution: process txs, keep state of blockchain up-to-date.

Storage?: maintain archive of entire sequence of processed txs.

Validator Responsibilities, Revisited

Consensus: agree on a sequence blocks/transactions.

Execution: process txs, keep state of blockchain up-to-date.

Storage?: maintain archive of entire sequence of processed txs.

Typical solution: validators expected to store what is necessary to efficiently check validity of txs and block (but nothing beyond this).

- at least a few entities (validators and/or third parties) maintain full history (last resort for reconstructing the blockchain state)

Validator Responsibilities, Revisited

Consensus: agree on a sequence blocks/transactions.

Execution: process txs, keep state of blockchain up-to-date.

Storage?: maintain archive of entire sequence of processed txs.

Typical solution: validators expected to store what is necessary to efficiently check validity of txs and block (but nothing beyond this).

- at least a few entities (validators and/or third parties) maintain full history (last resort for reconstructing the blockchain state)

Data availability (DA) problem: how to be sure that data really is being stored as intended?

Tendermint with Blockhashes

Optimizations to the original Tendermint protocol:

Tendermint with Blockhashes

Optimizations to the original Tendermint protocol:

1. leader proposes block (w/predecessor specified by hash) rather than a full blockchain

Tendermint with Blockhashes

Optimizations to the original Tendermint protocol:

1. leader proposes block (w/predecessor specified by hash) rather than a full blockchain
2. validators send “up-to-date” message only if they’ve seen all relevant data (predecessors), include only blockhash in msg

Tendermint with Blockhashes

Optimizations to the original Tendermint protocol:

1. leader proposes block (w/predecessor specified by hash) rather than a full blockchain
2. validators send "up-to-date" message only if they've seen all relevant data (predecessors), include only blockhash in msg
3. validators send "ack" message upon hearing > 2n/3 "up-to-date" messages for the same blockhash
   – even if haven't yet received the corresponding block

Tendermint with Blockhashes

Optimizations to the original Tendermint protocol:

1. leader proposes block (w/predecessor specified by hash) rather than a full blockchain
2. validators send “up-to-date” message only if they’ve seen all relevant data (predecessors), include only blockhash in msg
3. validators send “ack” message upon hearing > 2n/3 “up-to-date” messages for the same blockhash
   - even if haven’t yet received the corresponding block

Question: why is optimization (3) safe?

Tendermint with Blockhashes

Optimizations to the original Tendermint protocol:

1. leader proposes block (w/predecessor specified by hash) rather than a full blockchain
2. validators send "up-to-date" message only if they've seen all relevant data (predecessors), include only blockhash in msg
3. validators send "ack" message upon hearing > 2n/3 "up-to-date" messages for the same blockhash (even if haven't yet received the corresponding block)

Question: why is optimization (3) safe?

– if receive > 2n/3 "up-to-date" msgs for same blockhash ➔ > n/3 of these from honest validators ➔ must have seen corresponding block

Tendermint with Blockhashes

Optimizations to the original Tendermint protocol:

1. leader proposes block (w/predecessor specified by hash) rather than a full blockchain
2. validators send “up-to-date” message only if they’ve seen all relevant data (predecessors), include only blockhash in msg
3. validators send “ack” message upon hearing > 2n/3 “up-to-date” messages for the same blockhash (even if haven’t yet received the corresponding block)

Question: why is optimization (3) safe?

– if receive > 2n/3 “up-to-date” msgs for same blockhash ➔ > n/3 of these from honest validators ➔ must have seen corresponding block

Implicit assumption: honest validators: (i) store entire blockchain; and (ii) upload past blocks to others on request.

Data Availability Committees (DACs)

- n servers (permissioned, with publicly known public keys)
- bound $f < n$ on how many servers might be faulty

Data Availability Committees (DACs)

- n servers (permissioned, with publicly known public keys)
- bound f < n on how many servers might be faulty

To archive data: send data x to all n servers, wait until ≥ f+1 servers send back signed “ack h(x)” messages. [h=hash fn]

Data Availability Committees (DACs)

- n servers (permissioned, with publicly known public keys)
- bound $f < n$ on how many servers might be faulty

To archive data: send data x to all n servers, wait until $\geq f+1$ servers send back signed "ack h(x)" messages. [h=hash fn]

Assumption: after signing an "ack h(x)" message, non-faulty server will store x and resend it on request, indefinitely.

Data Availability Committees (DACs)

- n servers (permissioned, with publicly known public keys)
- bound $f < n$ on how many servers might be faulty

To archive data: send data x to all n servers, wait until $\geq f+1$ servers send back signed "ack h(x)" messages. [h=hash fn]

Assumption: after signing an "ack h(x)" message, non-faulty server will store x and resend it on request, indefinitely.

– any collection of $f+1$ signatures acking h(x) is proof (modulo assumptions) that the data x is available

Data Availability Committees (DACs)

- n servers (permissioned, with publicly known public keys)
- bound $f < n$ on how many servers might be faulty

To archive data: send data x to all n servers, wait until $\geq f+1$ servers send back signed "ack h(x)" messages. [h=hash fn]

Assumption: after signing an "ack h(x)" message, non-faulty server will store x and resend it on request, indefinitely.

- any collection of f+1 signatures acking h(x) is proof (modulo assumptions) that the data x is available
- example: Arbitrum AnyTrust (e.g., n=12, f=10)

Verifiable Information Dispersal (VID)

Setup: want to store data values $m_0, m_1, m_2, \ldots, m_{k-1}$. (e.g., 256 bits each)

– split data into chunks, if necessary

Verifiable Information Dispersal (VID)

Setup: want to store data values $m_0,m_1,m_2,\ldots,m_{k-1}$. (e.g., 256 bits each)

– split data into chunks, if necessary

Naïve DAC: send all data to each of n servers, factor-n overhead.

Verifiable Information Dispersal (VID)

Setup: want to store data values $m_0,m_1,m_2,\ldots,m_{k-1}$. (e.g., 256 bits each)

– split data into chunks, if necessary

Naïve DAC: send all data to each of n servers, factor-n overhead.

Idea: send each m_i to only one (or a few) servers.

Verifiable Information Dispersal (VID)

Setup: want to store data values $m_0, m_1, m_2, \ldots, m_{k-1}$. (e.g., 256 bits each)

– split data into chunks, if necessary

Naïve DAC: send all data to each of n servers, factor-n overhead.

Idea: send each m_i to only one (or a few) servers.

Issue: if sent only to faulty servers, could get lost.

Verifiable Information Dispersal (VID)

Setup: want to store data values $m_0,m_1,m_2,\ldots,m_{k-1}$. (e.g., 256 bits each)

– split data into chunks, if necessary

Naïve DAC: send all data to each of n servers, factor-n overhead.

Idea: send each m_i to only one (or a few) servers.

Issue: if sent only to faulty servers, could get lost.

Solution: add redundancy to the m_i's using an erasure code.

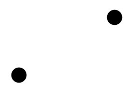
– in effect, every server will store a little bit of info about every m_i

Reed-Solomon Codes

Recall: any two points determine a (unique) line.

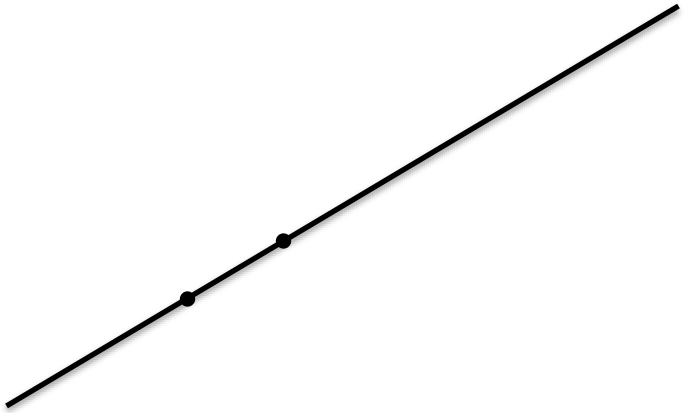
Reed-Solomon Codes

Recall: any two points determine a (unique) line.



Reed-Solomon Codes

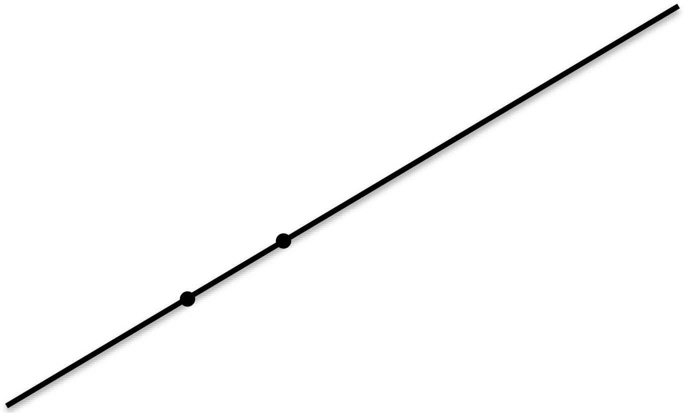
Recall: any two points determine a (unique) line.



Reed-Solomon Codes

Recall: any two points determine a (unique) line.

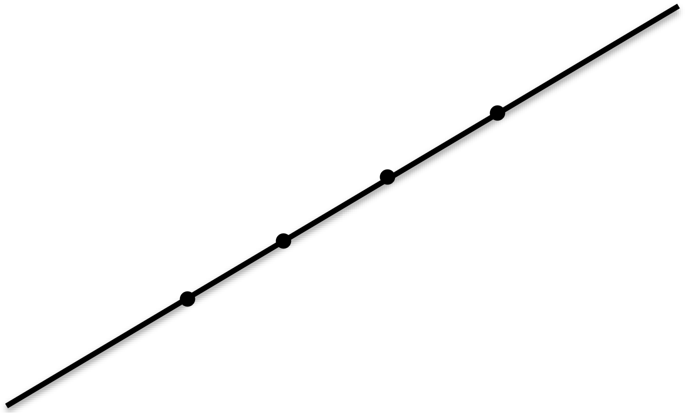
- example: points (1,4) and (2,9) ➔ line y = 5x-1



Reed-Solomon Codes

Recall: any two points determine a (unique) line.

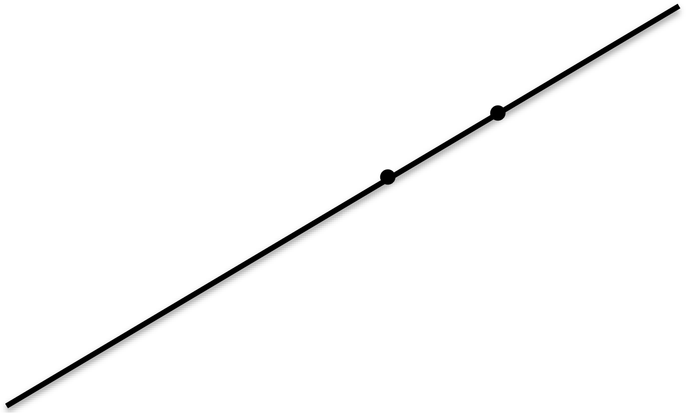
- example: points (1,4) and (2,9) ➔ line y = 5x-1
- note: could redundantly encode line y = 5x -1 via the points (1,4), (2,9), (3,14), and (4,19)



Reed-Solomon Codes

Recall: any two points determine a (unique) line.

- example: points (1,4) and (2,9) ➔ line y = 5x-1
- note: could redundantly encode line y = 5x -1 via the points (1,4), (2,9), (3,14), and (4,19)
  - – if later remember only (3,14) and (4,19), can still recover line 5x-1



Reed-Solomon Codes

Recall: any two points determine a (unique) line.

- note: could redundantly encode line $y = 5x - 1$ via the points (1,4), (2,9), (3,14), and (4,19)

Reed-Solomon Codes

Recall: any two points determine a (unique) line.

- note: could redundantly encode line y = 5x -1 via the points (1,4), (2,9), (3,14), and (4,19)

Recall: a (degree-d) polynomial has the form

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{d} a_i x^i.$$

Reed-Solomon Codes

Recall: any two points determine a (unique) line.

- note: could redundantly encode line y = 5x -1 via the points (1,4), (2,9), (3,14), and (4,19)

Recall: a (degree-d) polynomial has the form

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{d} a_i x^i.$$

Fact: any d+1 points $(x_0,y_0),(x_1,y_1),\ldots,(x_d,y_d)$ determine a unique degree-d polynomial, f(x) [with $y_i=f(x_i)$ for all i=0,1,2,...,d].

- if store n > d+1 evaluations of f, can recover f from any d+1 of them

Reed-Solomon Codes

Recall: any two points determine a (unique) line.

- note: could redundantly encode line y = 5x -1 via the points (1,4), (2,9), (3,14), and (4,19)

Recall: a (degree-d) polynomial has the form

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{d} a_i x^i .$$

Fact: any d+1 points $(x_0,y_0),(x_1,y_1),\ldots,(x_d,y_d)$ determine a unique degree-d polynomial, f(x) [with $y_i=f(x_i)$ for all i=0,1,2,...,d].

- if store n > d+1 evaluations of f, can recover f from any d+1 of them
- easy to compute f from evaluations, e.g. via Lagrange interpolation
- corresponds to a *Reed-Solomon code* with parameters n and k=d+1

The Cachin-Tessaro VID Protocol

Setup: sender with k 256-bit values, $m_0, m_1, m_2, \ldots, m_{k-1}$; n servers.

The Cachin-Tessaro VID Protocol

Setup: sender with k 256-bit values, $m_0,m_1,m_2,\ldots,m_{k-1}$; n servers.

Assumption: at most f faulty servers, with $f < n/3$ and $n-f \geq k$.

– ex: $n=3k+1$, $f=k$

The Cachin-Tessaro VID Protocol

Setup: sender with k 256-bit values, $m_0, m_1, m_2, \ldots, m_{k-1}$; n servers.

Assumption: at most f faulty servers, with $f < n/3$ and $n-f \geq k$.

- ex: $n=3k+1$, $f=k$

Step 1: sender interprets m_i's as integers modulo p (p prime, $\approx 2^{256}$)

The Cachin-Tessaro VID Protocol

Setup: sender with k 256-bit values, $m_0, m_1, m_2, \ldots, m_{k-1}$; n servers.

Assumption: at most f faulty servers, with $f < n/3$ and $n-f \geq k$.

- ex: $n=3k+1$, $f=k$

Step 1: sender interprets m_i's as integers modulo p (p prime, $\approx 2^{256}$)

- sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod{p}$

The Cachin-Tessaro VID Protocol

Setup: sender with k 256-bit values, $m_0, m_1, m_2, \ldots, m_{k-1}$; n servers.

Assumption: at most f faulty servers, with $f < n/3$ and $n-f \geq k$.

- ex: $n=3k+1$, $f=k$

Step 1: sender interprets m_i's as integers modulo p (p prime, $\approx 2^{256}$)

- sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod{p}$
- sender computes $y_i = f(i)$ for $i=1,2,\ldots,n$

The Cachin-Tessaro VID Protocol

Setup: sender with k 256-bit values, $m_0, m_1, m_2, \ldots, m_{k-1}$; n servers.

Assumption: at most f faulty servers, with $f < n/3$ and $n-f \geq k$.

- ex: $n=3k+1$, $f=k$

Step 1: sender interprets m_i's as integers modulo p (p prime, $\approx 2^{256}$)

- sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \ (mod\ p)$
- sender computes $y_i = f(i)$ for $i=1,2,\ldots,n$
- sender computes Merkle tree T with y_i's as leaves

The Cachin-Tessaro VID Protocol

Setup: sender with k 256-bit values, $m_0,m_1,m_2,\ldots,m_{k-1}$; n servers.

Assumption: at most f faulty servers, with $f < n/3$ and $n-f \geq k$.

- ex: $n=3k+1$, $f=k$

Step 1: sender interprets m_i's as integers modulo p (p prime, $\approx 2^{256}$)

- sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod{p}$
- sender computes $y_i = f(i)$ for $i=1,2,\ldots,n$
- sender computes Merkle tree T with y_i's as leaves
- sender sends (r, y_i, π_i) to server i [r = Merkle root, π_i = Merkle proof]

The Cachin-Tessaro VID Protocol (con'd)

Step 1: sender interprets m_i's as integers modulo p (p prime, ≈2^{256})

- sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod p$
- sender computes y_i = f(i) for i=1,2,…,n
- sender computes Merkle tree T with y_i's as leaves
- sender sends (r,y_i,π_i) to server i [r = Merkle root, π_i = Merkle proof]

The Cachin-Tessaro VID Protocol (con’d)

Step 1: sender interprets m_i’s as integers modulo p (p prime, ≈2^{256})

- sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod{p}$
- sender computes y_i = f(i) for i=1,2,…,n
- sender computes Merkle tree T with y_i’s as leaves
- sender sends (r, y_i, π_i) to server i [r = Merkle root, π_i = Merkle proof]

Step 2: [if sender might equivocate] servers reach consensus on r

The Cachin-Tessaro VID Protocol (con'd)

Step 1: sender interprets m_i's as integers modulo p (p prime, ≈2^{256})

- sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod p$
- sender computes y_i = f(i) for i=1,2,…,n
- sender computes Merkle tree T with y_i's as leaves
- sender sends (r,y_i,π_i) to server i [r = Merkle root, π_i = Merkle proof]

Step 2: [if sender might equivocate] servers reach consensus on r

- if server i received valid tuple (r,y_i,π_i) from sender, send back an "ACK r" message (with server's signature)

The Cachin-Tessaro VID Protocol (con'd)

Step 1: sender interprets m_i's as integers modulo p (p prime, $\approx 2^{256}$)

- sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod p$
- sender computes $y_i = f(i)$ for i=1,2,…,n
- sender computes Merkle tree T with y_i's as leaves
- sender sends (r, y_i, π_i) to server i [r = Merkle root, π_i = Merkle proof]

Step 2: [if sender might equivocate] servers reach consensus on r

- if server i received valid tuple (r, y_i, π_i) from sender, send back an "ACK r" message (with server's signature)

Step 3: if sender receives ≥ k+f signed "ACK r" messages, assemble into a "DA certificate."

The Cachin-Tessaro VID Protocol (con'd)

Step 1: sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod p$

- sender computes $y_i = f(i)$ for i=1,2,…,n, computes Merkle tree T with y_i's as leaves
- sender sends (r, y_i, π_i) to server i [r = Merkle root, π_i = Merkle proof]

Step 2: if server i received valid tuple (r, y_i, π_i) from sender, send back an "ACK r" message

Step 3: if sender receives ≥ k+f signed "ACK r" messages, assemble into a "DA certificate."

To reconstruct: (given a DA certificate for data $m_0, m_1, m_2, \ldots, m_{k-1}$)

The Cachin-Tessaro VID Protocol (con'd)

Step 1: sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \pmod{p}$

- sender computes $y_i = f(i)$ for i=1,2,…,n, computes Merkle tree T with y_i's as leaves
- sender sends (r, y_i, π_i) to server i [r = Merkle root, π_i = Merkle proof]

Step 2: if server i received valid tuple (r, y_i, π_i) from sender, send back an "ACK r" message

Step 3: if sender receives ≥ k+f signed "ACK r" messages, assemble into a "DA certificate."

To reconstruct: (given a DA certificate for data $m_0, m_1, m_2, \ldots, m_{k-1}$)

– collect a set S of k signed tuples of the form (r, y_i, π_i)

The Cachin-Tessaro VID Protocol (con'd)

Step 1: sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \ (mod\, p)$

- sender computes y_i = f(i) for i=1,2,…,n, computes Merkle tree T with y_i's as leaves
- sender sends (r,y_i,π_i) to server i [r = Merkle root, π_i = Merkle proof]

Step 2: if server i received valid tuple (r,y_i,π_i) from sender, send back an "ACK r" message

Step 3: if sender receives ≥ k+f signed "ACK r" messages, assemble into a "DA certificate."

To reconstruct: (given a DA certificate for data m_0,m_1,m_2,…,m_{k-1})

- collect a set S of k signed tuples of the form (r,y_i,π_i)
- compute degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} a_i x^i$ s.t. f(i)=y_i for all i in S

The Cachin-Tessaro VID Protocol (con'd)

Step 1: sender forms degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} m_i x^i \ (mod\ p)$

- sender computes y_i = f(i) for i=1,2,…,n, computes Merkle tree T with y_i's as leaves
- sender sends (r,y_i,π_i) to server i [r = Merkle root, π_i = Merkle proof]

Step 2: if server i received valid tuple (r,y_i,π_i) from sender, send back an "ACK r" message

Step 3: if sender receives ≥ k+f signed "ACK r" messages, assemble into a "DA certificate."

To reconstruct: (given a DA certificate for data m_0,m_1,m_2,…,m_{k-1})

– collect a set S of k signed tuples of the form (r,y_i,π_i)

– compute degree-(k-1) polynomial $f(x) = \sum_{i=0}^{k-1} a_i x^i$ s.t. f(i)=y_i for all i in S

– compute Merkle tree with values f(1),f(2),…,f(n) at leaves, check root = r

  - if not, sender committed to polynomial w/degree > k-1 (fail)

Data Availability Sampling (DAS)

Question: as an end user, how to be sure that a collection of servers really is storing critical data?

- e.g., don’t accept the trust assumptions required by a DA certificate

Data Availability Sampling (DAS)

Question: as an end user, how to be sure that a collection of servers really is storing critical data?

– e.g., don't accept the trust assumptions required by a DA certificate

Naïve solution: download data from servers to double-check.

– what if this is infeasible? (too much data and/or weak device)

Data Availability Sampling (DAS)

Question: as an end user, how to be sure that a collection of servers really is storing critical data?

- e.g., don’t accept the trust assumptions required by a DA certificate

Naïve solution: download data from servers to double-check.

- what if this is infeasible? (too much data and/or weak device)

Idea: download several random *chunks* of data to audit.

Data Availability Sampling (DAS)

Question: as end user, how to be sure that servers are storing critical data?

Idea: download several random *chunks* of data to audit.

Intuition: suppose crash failures (of servers) only.

Data Availability Sampling (DAS)

Question: as end user, how to be sure that servers are storing critical data?

Idea: download several random *chunks* of data to audit.

Intuition: suppose crash failures (of servers) only.

- chunk data into k pieces, use a Reed-Solomon code to redundantly encode as 2k chunks (as in VID protocol), store chunks with servers

Data Availability Sampling (DAS)

Question: as end user, how to be sure that servers are storing critical data?

Idea: download several random *chunks* of data to audit.

Intuition: suppose crash failures (of servers) only.

- chunk data into k pieces, use a Reed-Solomon code to redundantly encode as 2k chunks (as in VID protocol), store chunks with servers
- if uncrashed servers still have ≥ k chunks ➔ data is still available (can reconstruct if needed)

Data Availability Sampling (DAS)

Question: as end user, how to be sure that servers are storing critical data?

Idea: download several random *chunks* of data to audit.

Intuition: suppose crash failures (of servers) only.

- chunk data into k pieces, use a Reed-Solomon code to redundantly encode as 2k chunks (as in VID protocol), store chunks with servers
- if uncrashed servers still have ≥ k chunks ➔ data is still available (can reconstruct if needed)
- if uncrashed servers only have < k chunks ➔ can recognize this with 99% probability by downloading 7 random chunks

Data Availability Sampling (DAS)

Question: as end user, how to be sure that servers are storing critical data?

Idea: download several random *chunks* of data to audit.

Intuition: suppose crash failures (of servers) only.

- chunk data into k pieces, use a Reed-Solomon code to redundantly encode as 2k chunks (as in VID protocol), store chunks with servers
- if uncrashed servers still have ≥ k chunks ➔ data is still available (can reconstruct if needed)
- if uncrashed servers only have < k chunks ➔ can recognize this with 99% probability by downloading 7 random chunks
- ideally, reconstruct chunks whenever they're detected as missing