

Samsara

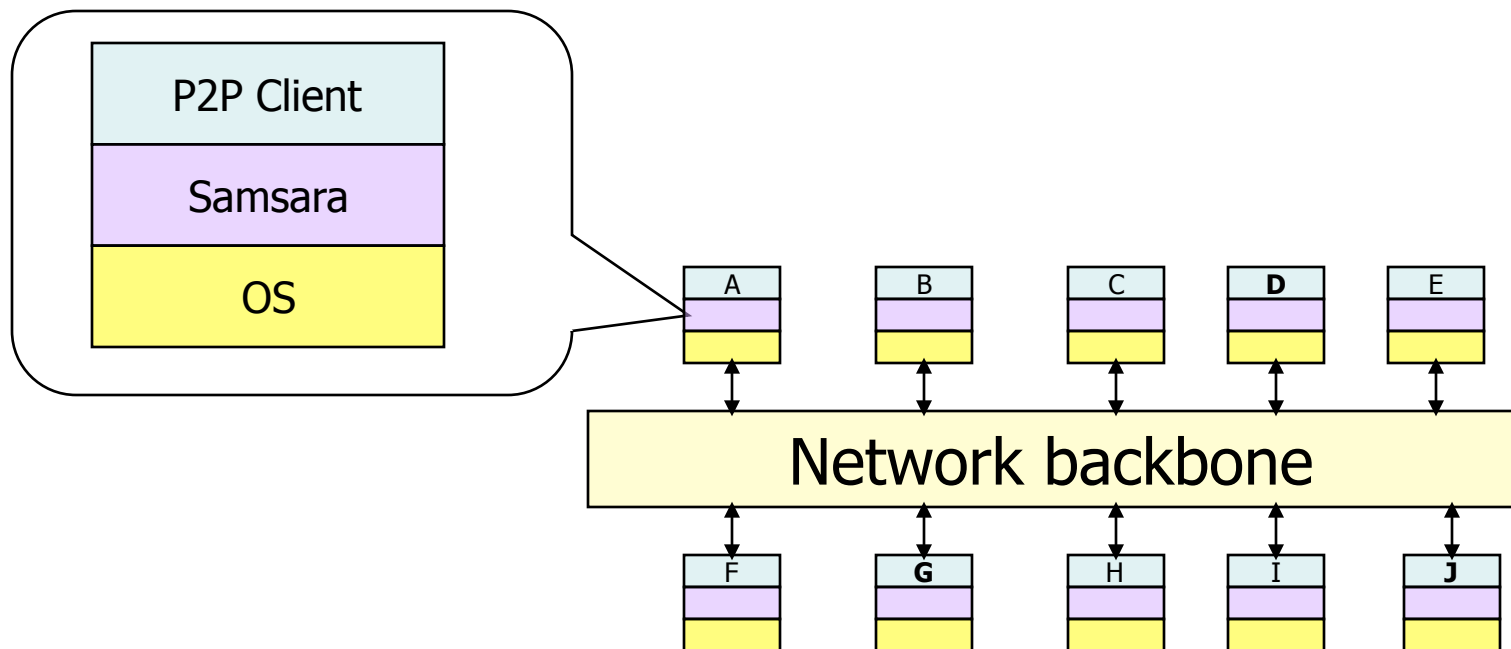
Honor among thieves in peer-to-peer storage

Objectives

- Storage system for peer-to-peer backup:
 - Consumption proportional to contribution
 - No centralized admin
 - punishment to cheaters,
 - Minimizing punishment to participants with transient failure

What is Samsara

- A storage system for peer-to-peer backup systems
- Controls the storage relationships for the P2P system



A,B,C...,J - Nodes in the P2P network

D,G,J... - Replica nodes of A

What is Samsara – *contd.*

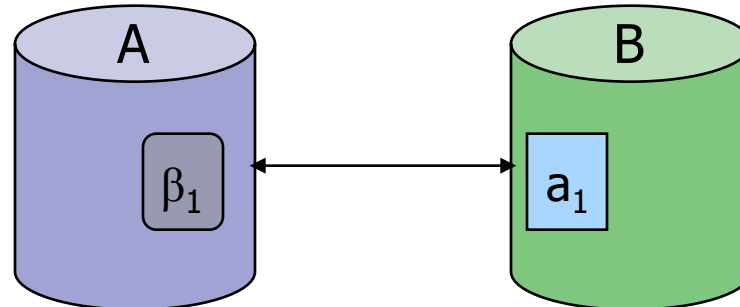
- A node can choose replica sites
- Every node maintains hash table of active content, helps in finding a better replica node
- The main functions of Samsara are –
 - Maintaining storage relationships
 - Creating symmetry
 - Punishing non-responsive nodes

Why is such a system needed?

- Problems with peer-to-peer storage systems –
 - Tragedy of the commons - Users have no incentive to contribute
 - Under-reporting of resources
- Problems with mechanisms to compel fairness
 - Trusted third parties require centralized administration
 - Certified identities and public keys require trusted means of certification
- Symmetrical systems are restrictive
 - No freedom in choice of replica sites
 - Transient failures are punished too severely

The Samsara model

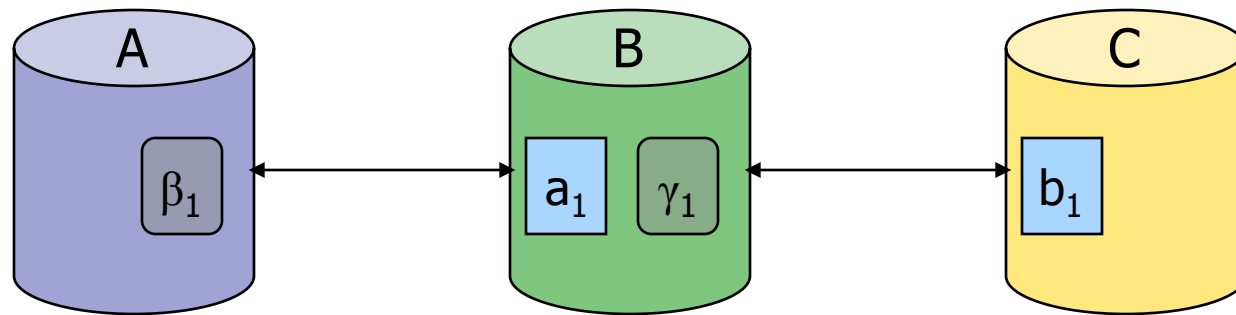
- No Greed - When you ask for space, promise same amount in return
- This promise is a **Claim**
- **Claims** are physical storage space reserved for the party that holds it



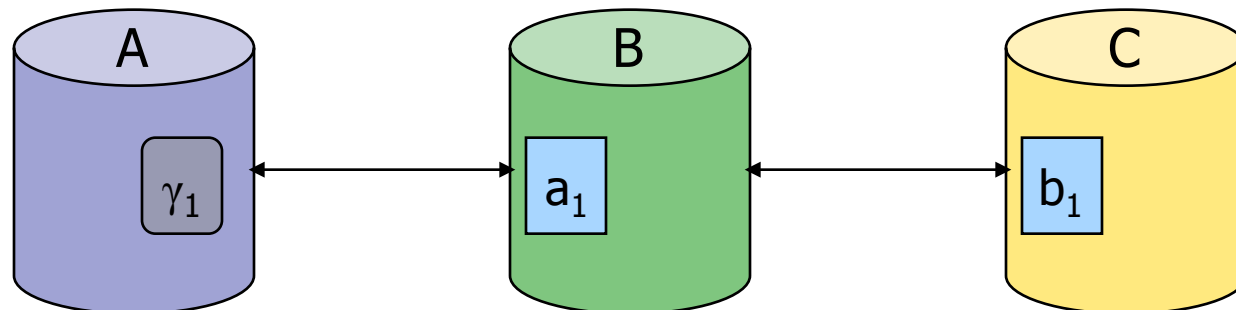
- A stores a_1 on B, and stores claim β_1 for B in return
- B has total ownership on its claim β_1 , it can use it to store data when it needs storage space

Claim forwarding

- Claims could be forwarded downstream



- B stores data for A, owns claim on A
- C stores data for B, owns claim on B
- B could forward C's claim to A in lieu of its own claim

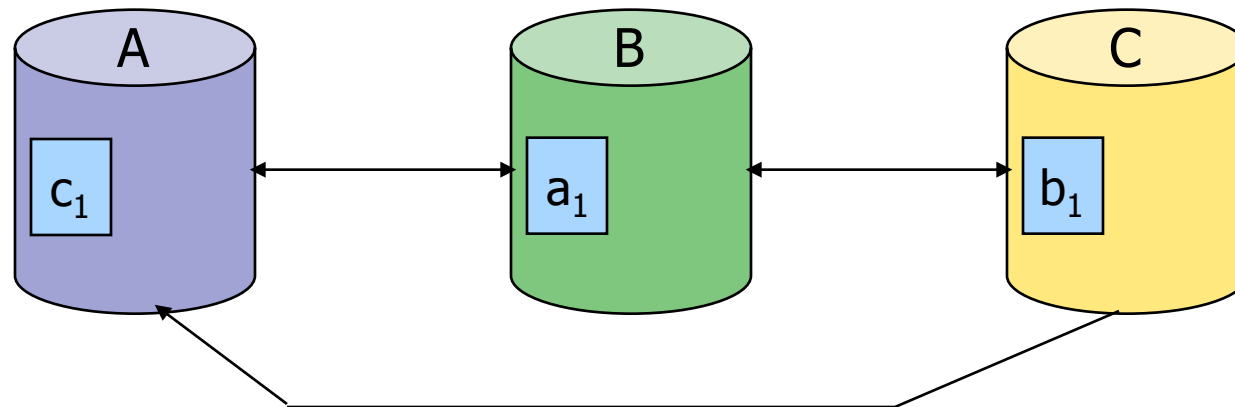


Claim forwarding – *contd.*

- Forwarded claim, not forwarded responsibility
 - A node still remains responsible for the claims it owes, even after forwarding
 - If a claim becomes unavailable then claim owner punishes the node it had the original claim on
 - Forwarding not preferable unless essential
 - The claim owner has information about the forwarding

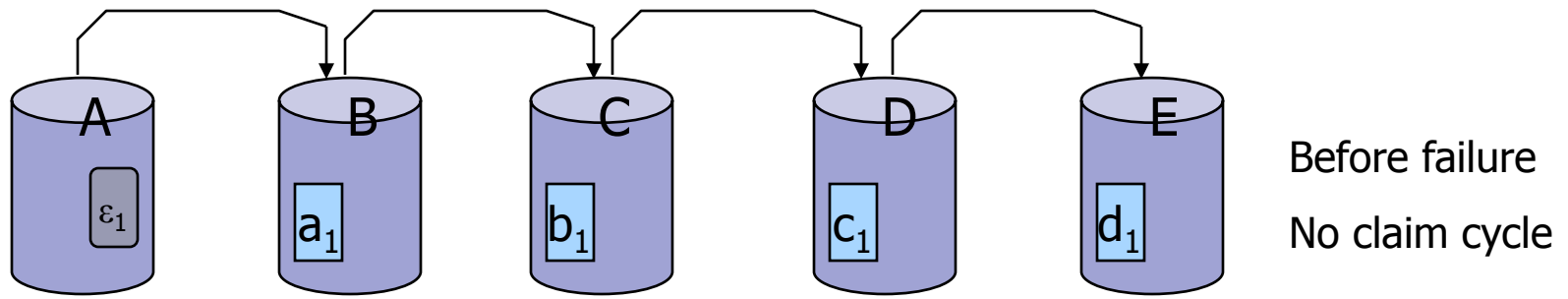
Claim cycle

- When a node wants space on some node that holds its forwarded claim
 - Continuing from the diagrams on slide 7

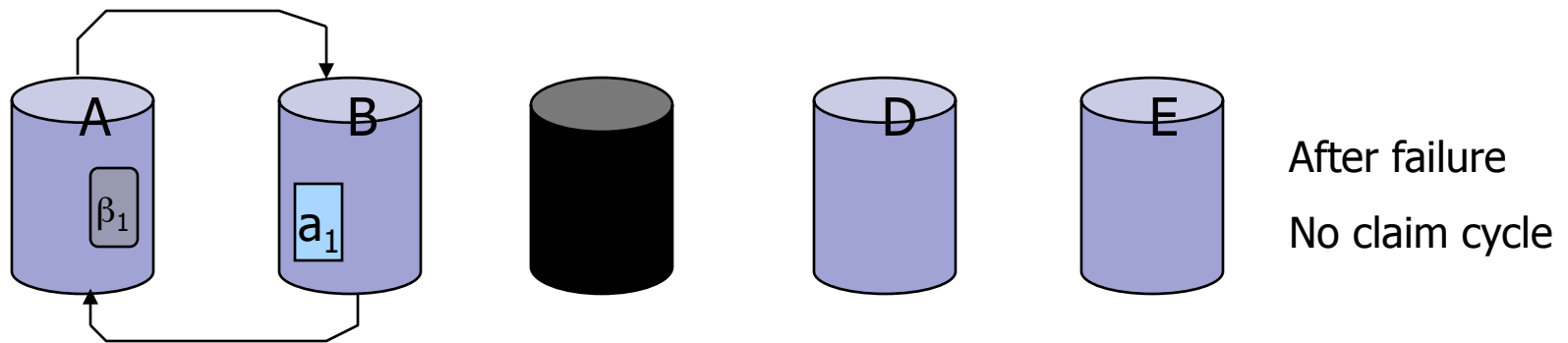


- C takes space on A, A passes C's claim back to C
- C deletes its claim

Reliability of forwarding

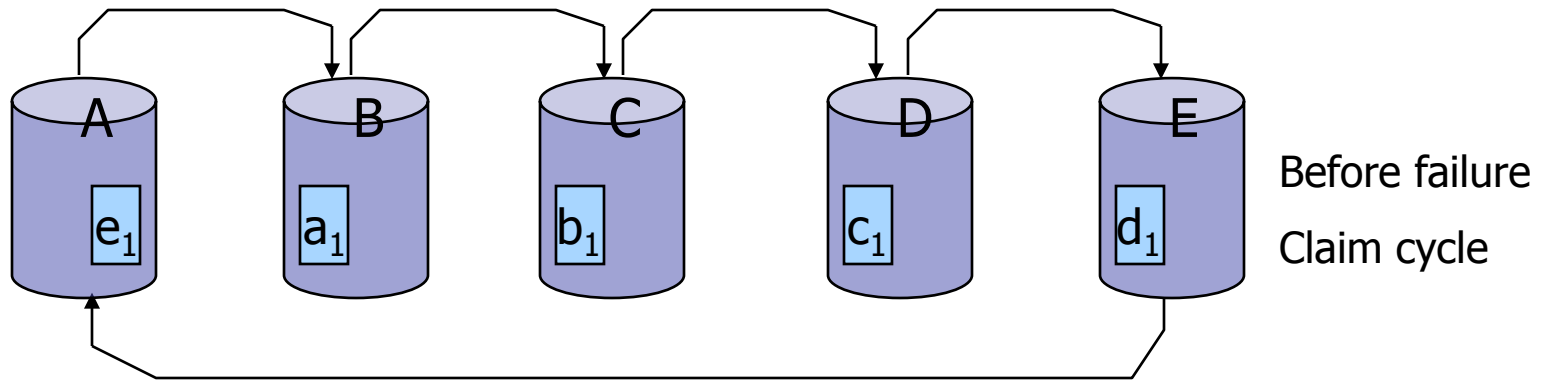


- If C fails

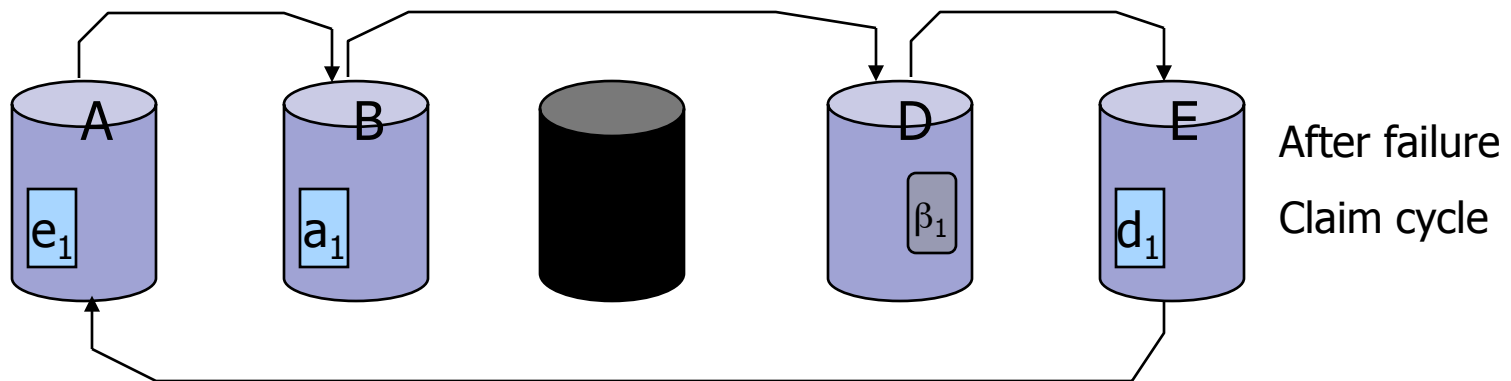


- All the data stored upstream of C is lost

Reliability of forwarding – *contd.*



- If C fails



- Only data stored on C is lost
- Claim cycles are more reliable

Claim construction

- Incompressible placeholders, provided in return of storage space
- Three values needed for computing claims
 - A secret pass-phrase P
 - A private, symmetric key K
 - A location in the storage space
- Process of claim computation
 - Claims are made of hash values
 - One hash value is 20 bytes long
 - i^{th} hash is SHA1 hash of concatenation of P and number i
 - $h_0 = \text{SHA1}(P, 0)$
 - $h_i = \text{SHA1}(P, i)$

Claim construction – *contd.*

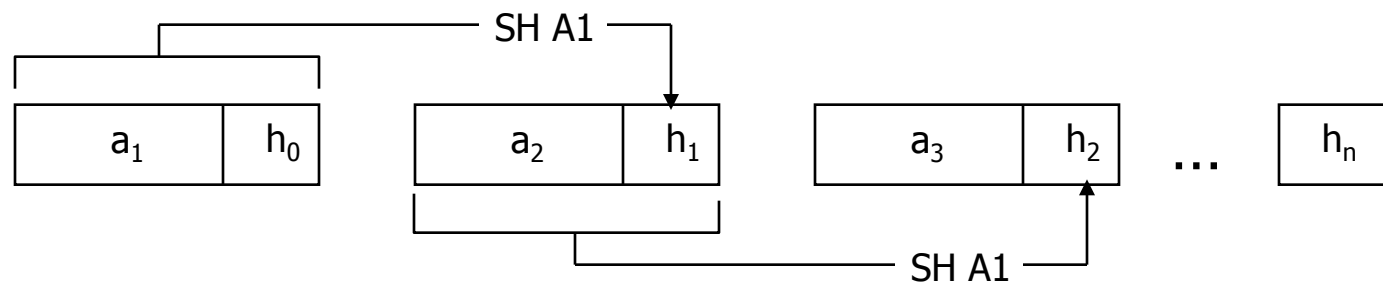
- Claims are fixed sized blocks
 - Formed from consecutive hash values
 - To construct 512 bytes long claims -
 - First claim C_0 = first 25 hashes + 12 bytes of 26th hash, then encrypting it with K
 - Claim $C_i = \{h_j, h_{j+1}, \dots, h_{j+24}, h_{j+25}[0], \dots, h_{j+25}[11]\}K$
Where $j = i * 26$

Querying Nodes

- Nodes need to monitor their remote storage
 - Need to check if the other nodes are keeping their part of the promise
- Need not be answered immediately
 - Querying node needs to be patient because –
 - The other node might be bogged down with some resource intensive process
 - The other node might be facing bandwidth shortage
- Need not be very frequent
 - More queries means more network cost for both the nodes
 - Querying every few hours or even once a day is enough

Querying Nodes – *contd.*

- Method of querying -
 - No need to return entire data object to prove data being held
 - Querying node –
 - sends a unique value, h_0 , along with list of n objects to be verified
 - Responding node –
 - Appends h_0 to first object in the list and computes SHA1 of this concatenation. This gives hash h_1
 - Appends h_1 to second object in the list and computes SHA1 of this concatenation to get h_2 and so on
 - After n^{th} object h_n is returned to the querying node
 - Querying node checks h_n to verify if that all objects are stored



Transient failure

- Cheating or transient failure
 - Need to distinguish between cheating nodes and nodes suffering from transient failure
 - No sure way of knowing between the two
 - Any node should not lose data for transient failure
 - Dishonest nodes need to be punished
 - Grace period is an option but could be too harsh on the failed nodes. Could be misused also

Transient failure – *contd.*

- Gradated grace period
 - A node gets sufficient grace period to respond
 - All the data is not lost after the grace period
 - Punishment gets more severe after every elapsing grace period
- Probabilistic punishment
 - For every failed query, a small part of responder's data is deleted by the querying node
 - The part of data object to be deleted is chosen probabilistically
 - Lost data could be reconstructed from the replica nodes
 - Probability of permanently losing part of data gets higher with every failed query
 - Cheating nodes will loose all the data eventually

Transient failure – *contd.*

- Chances of misuse
 - A cheating node could have too many replicas or could create brand new set of replicas
 - Will have to do this very frequently
 - Network cost more than storage cost
 - For large amount of data it not economical to cheat
 - Smaller the amount of data, higher the success rate of cheating

Advantages & disadvantages

- Advantages
 - Tackles the issue of unchecked consumption
 - Provides flexibility in form of claim forwarding
 - Doesn't need centralized administration
 - Tries not to punish the nodes experiencing transient failure while punishing the dishonest users
 - Ensures compliance with minimum network load
- Disadvantages
 - A chain of forwarded claims can fail because of one bad node

Thoughts

- Claims are absolutely meaningless unless:
 - Symmetry
 - Forwarding
 - But they discourage forwarding!!
- No incentive to accept data unless need to store
- No penalty for refusing
- No time-shifting

Distributed Trust Closures in NICE

Seungjoon Lee

Rob Sherwood

Bobby Bhattacharjee

University of Maryland



www.cs.umd.edu/projects/nice

Cooperative Application Models

- Centralized trust repository (Mojonation, ebay)
 - requires central registry and authority

Cooperative Application Models

- Centralized trust repository (Mojonation, ebay)
 - requires central registry and authority
- Implicit cooperation (p2p apps)
 - *any* individual user may choose not to cooperate
 - but they still get full benefits from the system
 - integrity and correctness depends on *all* users cooperating
 - Current solutions don't work well, e.g. quotas in CFS, or require centralized certificate authorities, e.g. secure routing in Pastry

Cooperative Application Models

- Centralized trust repository (Mojonation, ebay)
 - requires central registry and authority
- Implicit cooperation (p2p apps)
 - *any* individual user may choose not to cooperate
but they still get full benefits from the system
 - integrity and correctness depends on *all* users cooperating
 - Current solutions don't work well, e.g. quotas in CFS, or require centralized certificate authorities, e.g. secure routing in Pastry
- **NICE: decentralized robust cooperation**
 - enable open cooperative applications

NICE: Preliminaries

- Each user chooses a NICE identifier
 - NICE id contains a public key
 - Key does not need to be published or certified
 - Users may simultaneously use multiple ids
- Each host has a owner
 - Owners set per-host policies
 - Host policies determine resource allocation and pricing

NICE in Operation

alice

bob

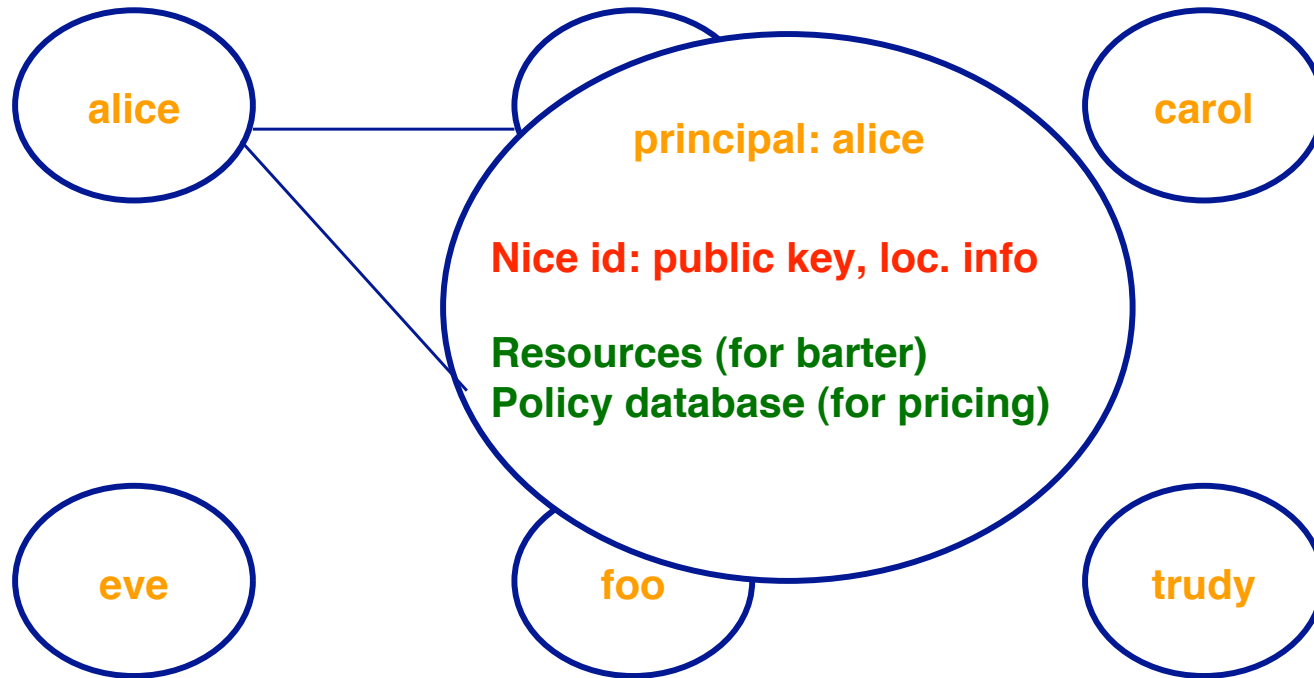
carol

eve

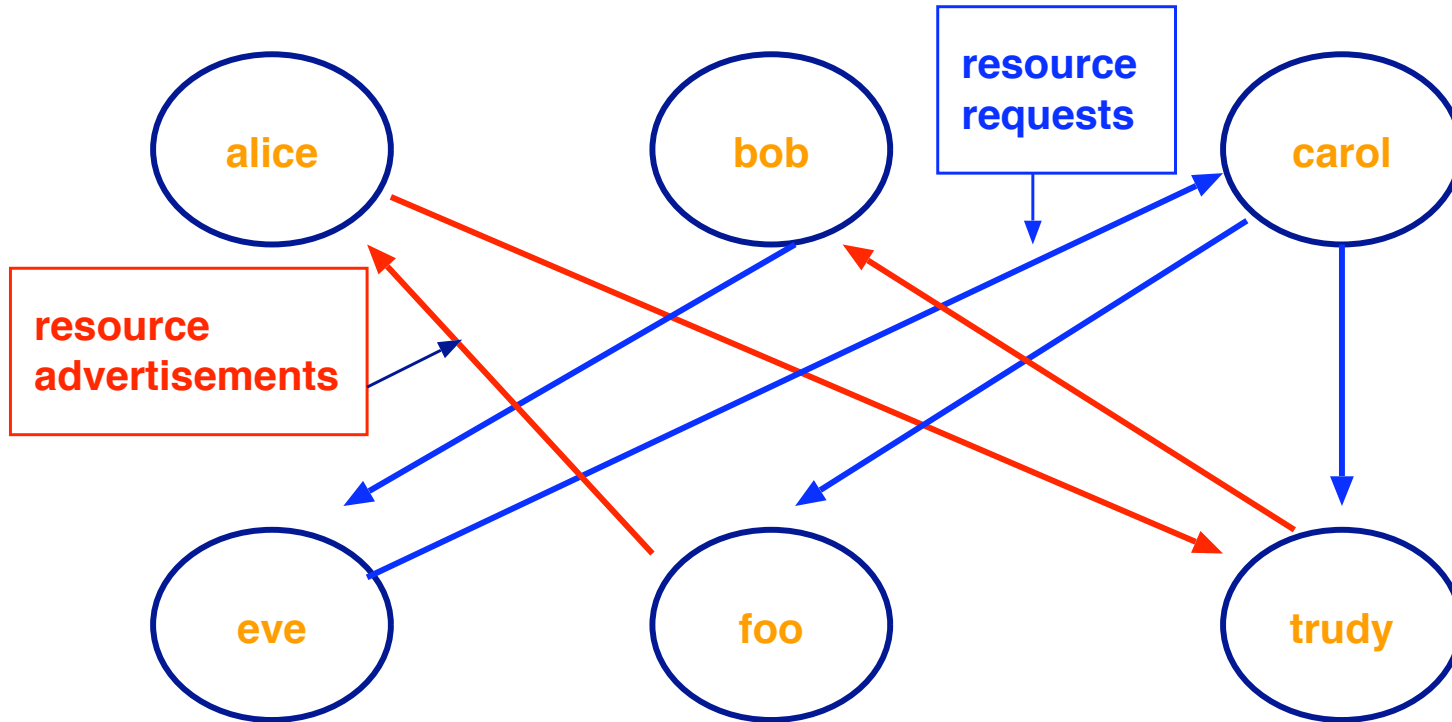
foo

trudy

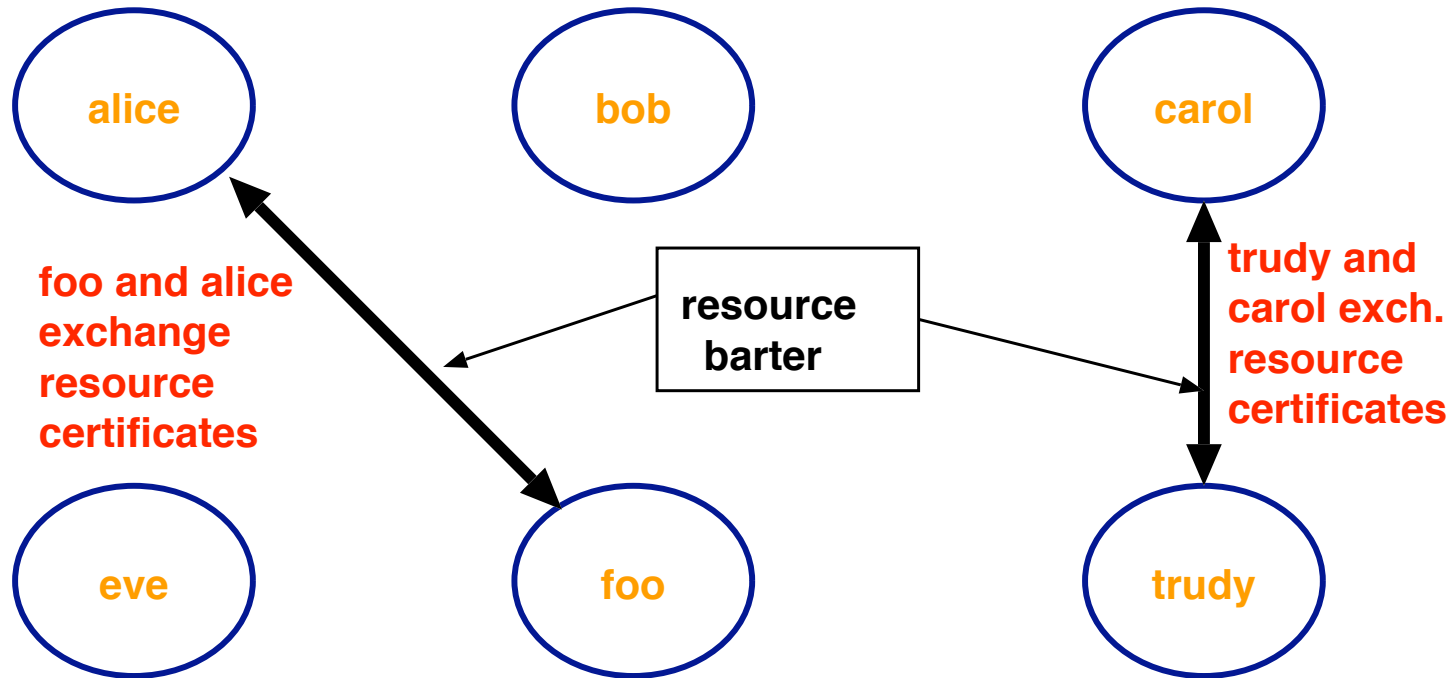
NICE in Operation



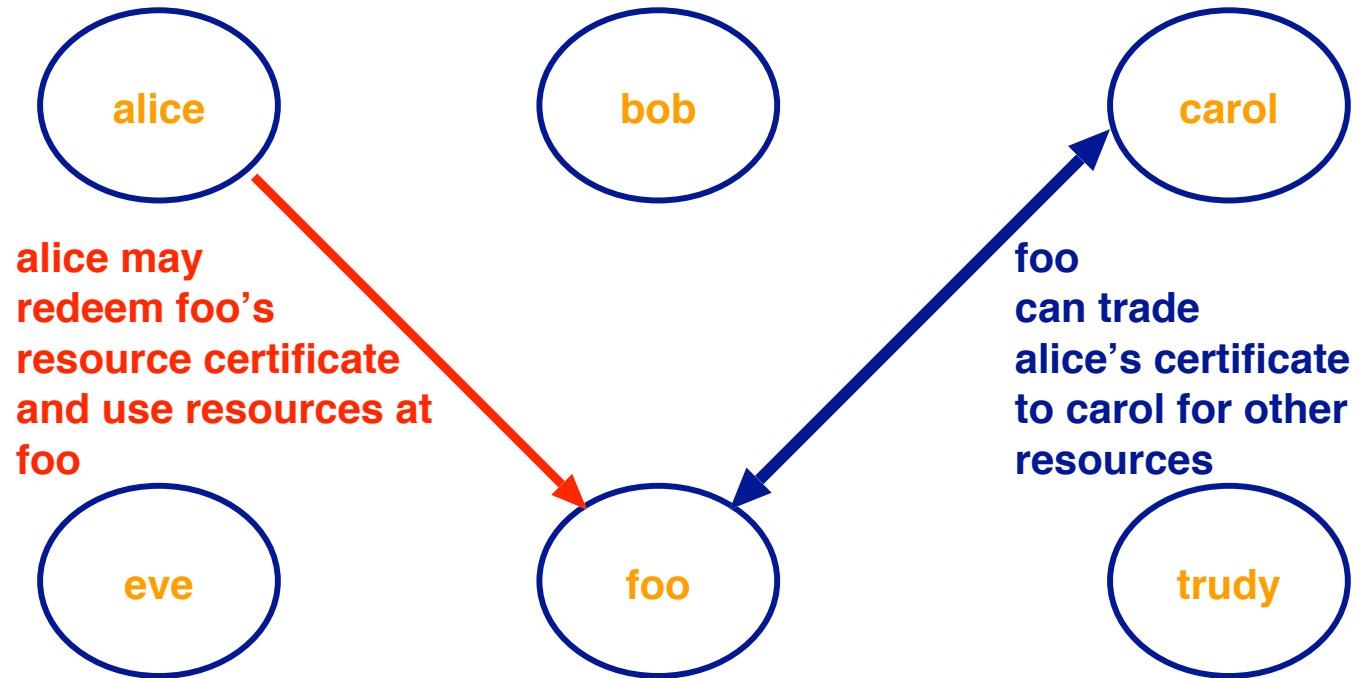
NICE in Operation



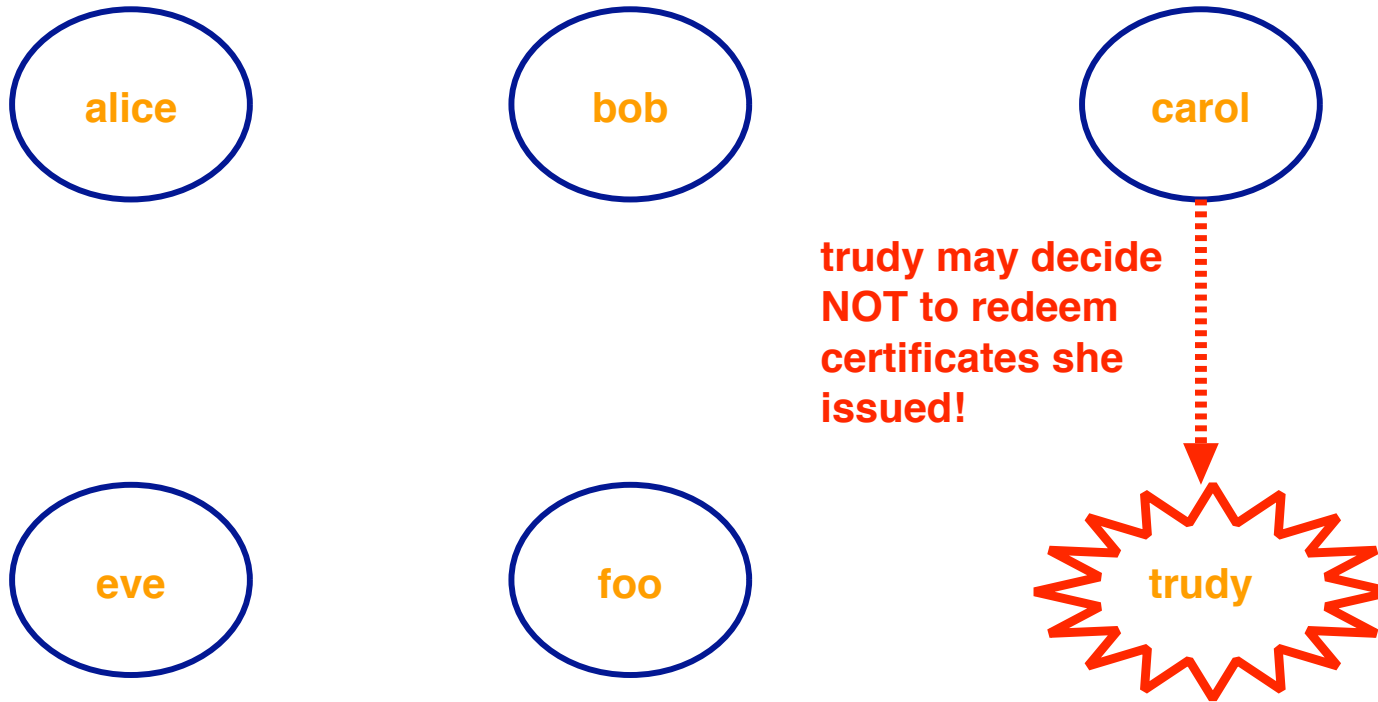
NICE in Operation



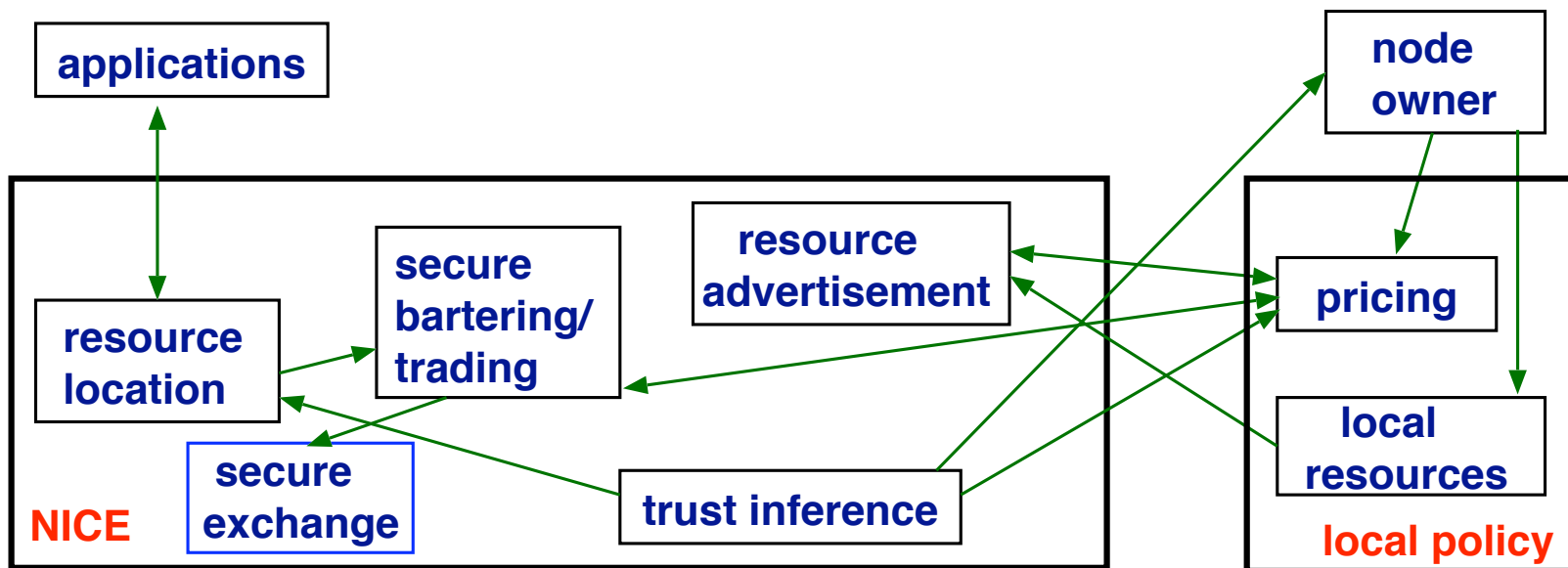
NICE in Operation



NICE in Operation



NICE: Component Architecture



- Users set local resource and pricing policy
- Applications request specific resources
- NICE locates appropriate resources and securely exchanges/trades *resource certificates*
- Resources certificates are redeemed for named resources

Resource Pricing

- Two main objective of default policies:
 - Form robust cooperative groups
 - Not lose large amounts of resources to malicious users
- Policies:
 - Trust-based pricing
 - Trust-based trading limits
- Default policies curb difficult DoS attacks

Trust Evaluation in NICE

- Integrity of entire NICE platform depends on trust computations
- $A \rightarrow B$ trust is a local measure (at A) of how likely A believes a transaction with B will be successful
- Users can use past experience to assign trust values
 - Trust can also be *inferred* through other trusted users

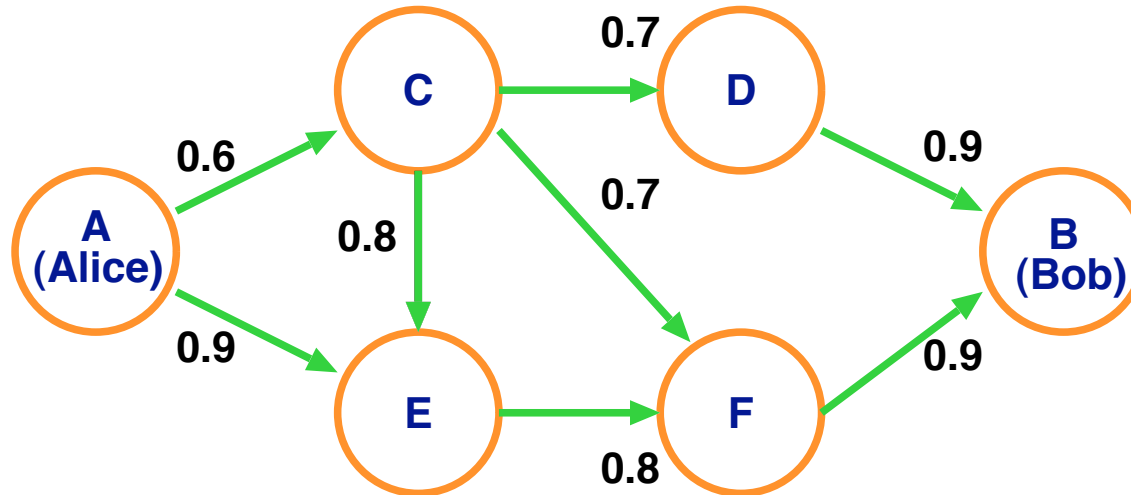
Goals of Trust Inference Schemes

- Users should be able to use local policy to assign trust
- Good nodes should *find* other good nodes efficiently . . .
... and not lose large amounts of resources
- Inference should be resilient against cooperating *malicious* groups
 - Malicious users disseminate arbitrary trust information
 - Good node cliques should be immune to such attacks

Centralized Trust Evaluation

- Trust Graph
 - Vertices are unique user identifiers
 - Directed edges represent how much source trusts dest.
- Many different inference algorithms feasible
 - Strongest Path
 - Weighted sum of disjoint strongest paths

Centralized Evaluation Examples



- Strongest Path: (AEFB, 0.8)
 - Inferred trust: 0.8
- Weighted sum of strongest disjoint paths:
 - Two disjoint paths: (AEFB, 0.8; wt. .9), (ACBD, 0.6; wt. .6)
 - Inferred trust: 0.72

Distributed Trust Inference

- Two main problems to distribute
 - Storage of trust information
 - Efficiently locate relevant edges
- If trust graph can be efficiently reproduced, then users can use any centralized algorithm to infer trust

Storing trust values in Cookies

- Suppose Alice redeems a resource certificate signed by Bob
- Alice assigns a $[0,1]$ value to the transaction *quality* and signs a transaction record — called a **cookie**
- **Bob** stores the cookie signed by Alice
 - Alice does *not* keep a record of the transaction!
 - The set of Alice's cookies stored by Bob determines the value of the Alice→Bob edge

Using cookies: base case

- Suppose Bob later wants to use Alice's resources:
 - Bob presents Alice a cookie(s) signed by Alice herself
 - Alice can verify her own signature . . .
... and use the cookie values to price her resources

Using Cookies: recursive case

- Bob wants to use Carol's resources but does not have cookies from Carol
 - Suppose Alice does have cookies from Carol
- Bob searches for Carol's cookies amongst users from whom he has cookies (Alice)
- Alice gives Bob a copy of the Carol→Alice cookies
- Bob presents Carol with a Carol→Bob **cookie path**
 - i.e. Bob produces a {Carol→Alice, Alice→Bob} cookie set
- Carol can now **infer** a trust value for Bob

Properties of cookie-based trust storage

- If Bob wants to use resources at Carol, *he* has to initiate a cookie search
 - Guards against a DoS attack
- Bob only stores statements of the form “X trusts Bob”
 - Clearly, Bob will discard any low valued cookies he gets
 - Users store cookies most beneficial to their own cause
- Transaction record storage is completely distributed
 - Fabricated transactions don't affect legitimate users

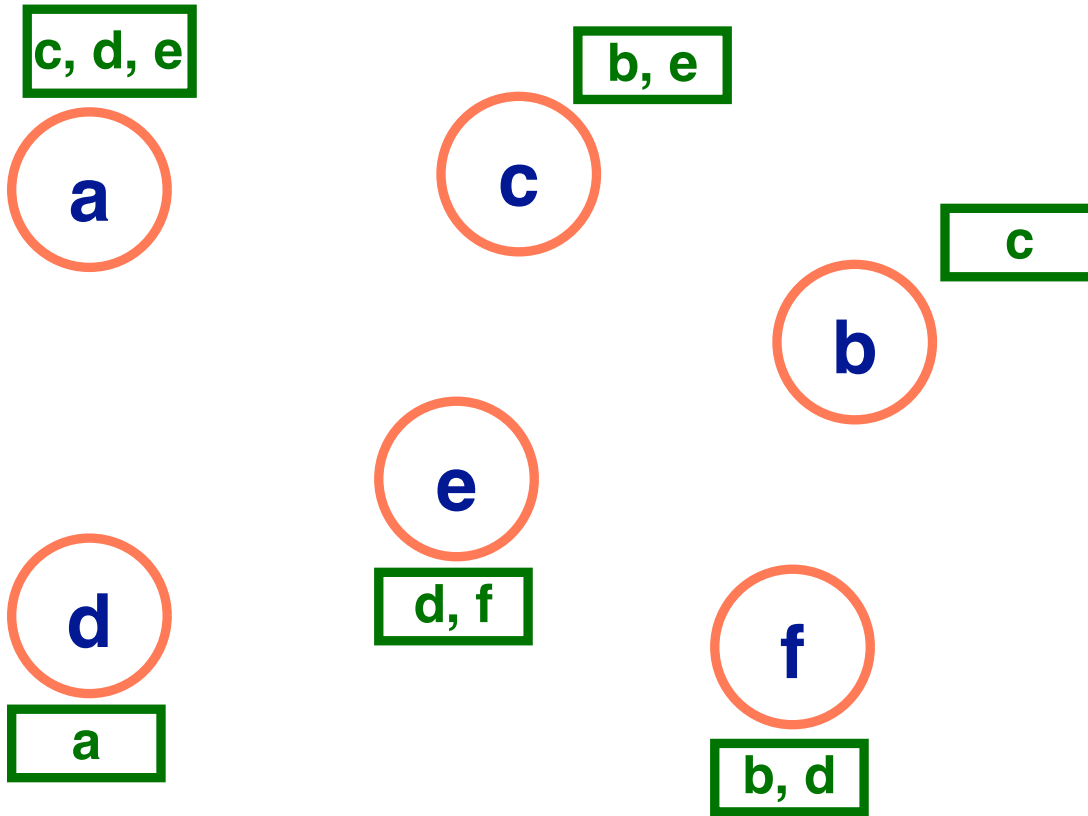
Properties of cookie-based trust storage

- If Bob wants to use resources at Carol, *he* has to initiate a cookie search
 - Guards against a DoS attack
- Bob only stores statements of the form “X trusts Bob”
 - Clearly, Bob will discard any low valued cookies he gets
 - Users store cookies most beneficial to their own cause
- Transaction record storage is completely distributed
 - Fabricated transactions don't affect legitimate users

How to locate cookies?

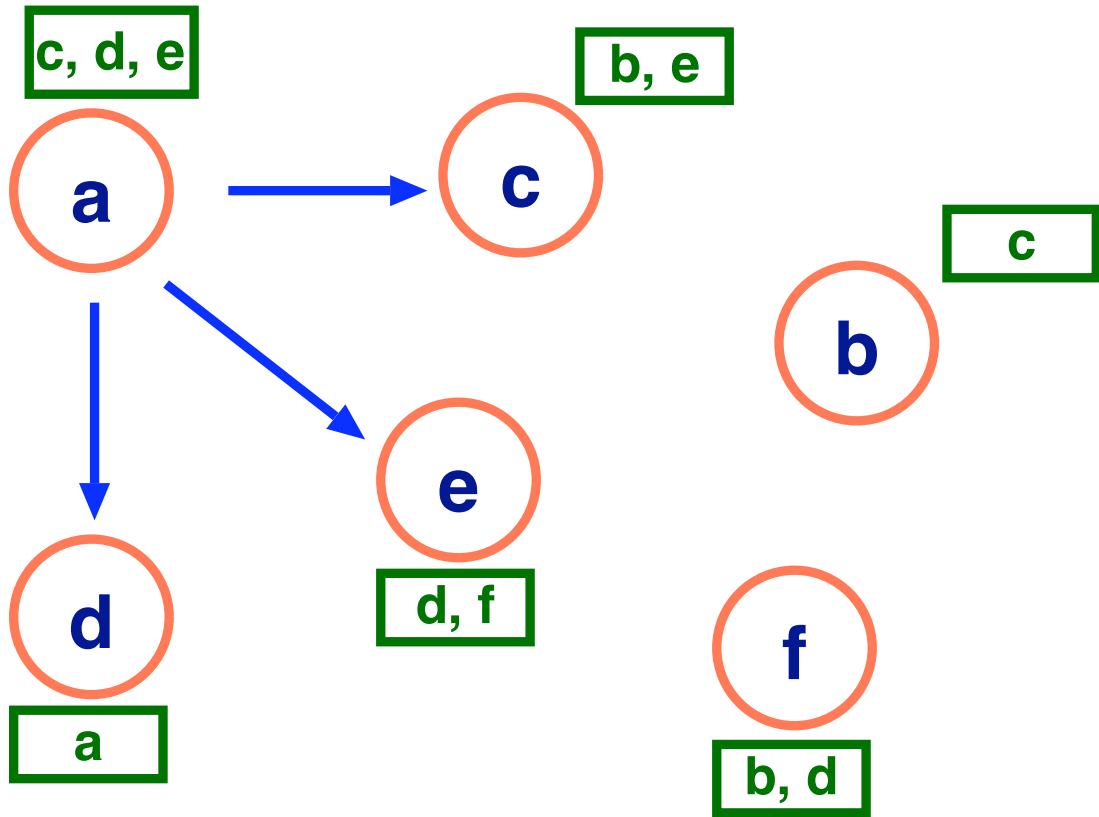
easy answer: flood queries for specific cookies

Flooding Example

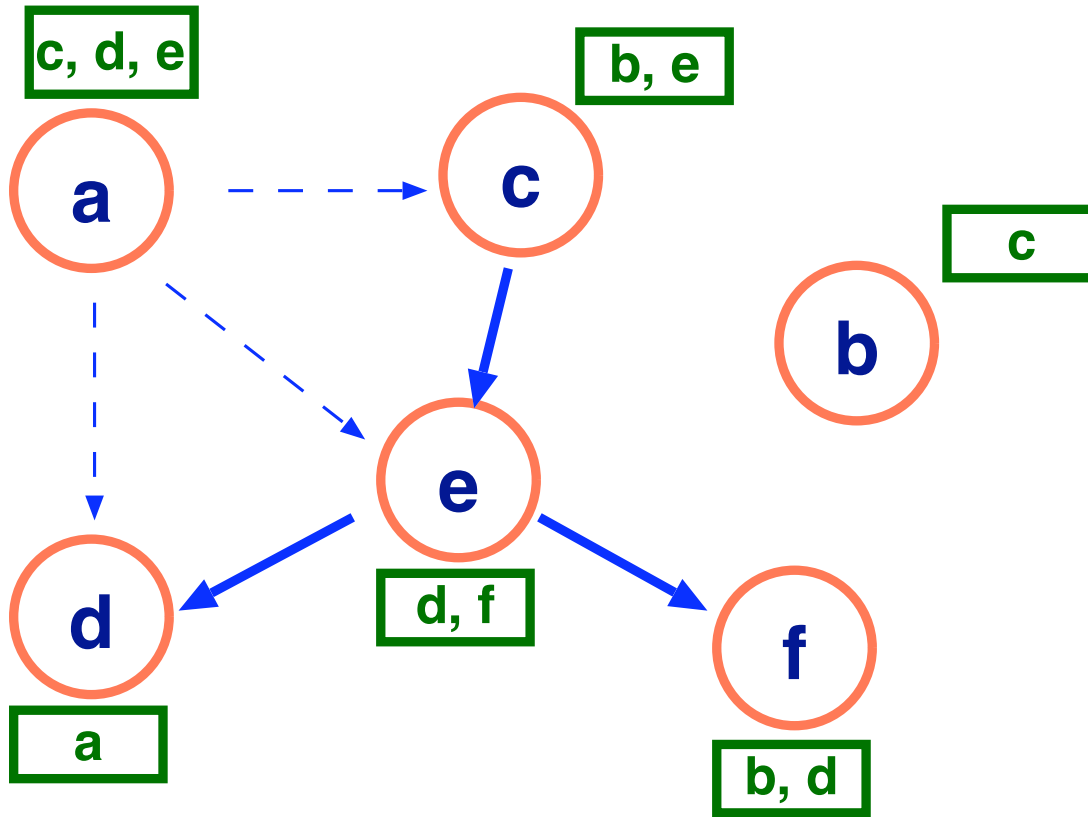


- Initial cookie state

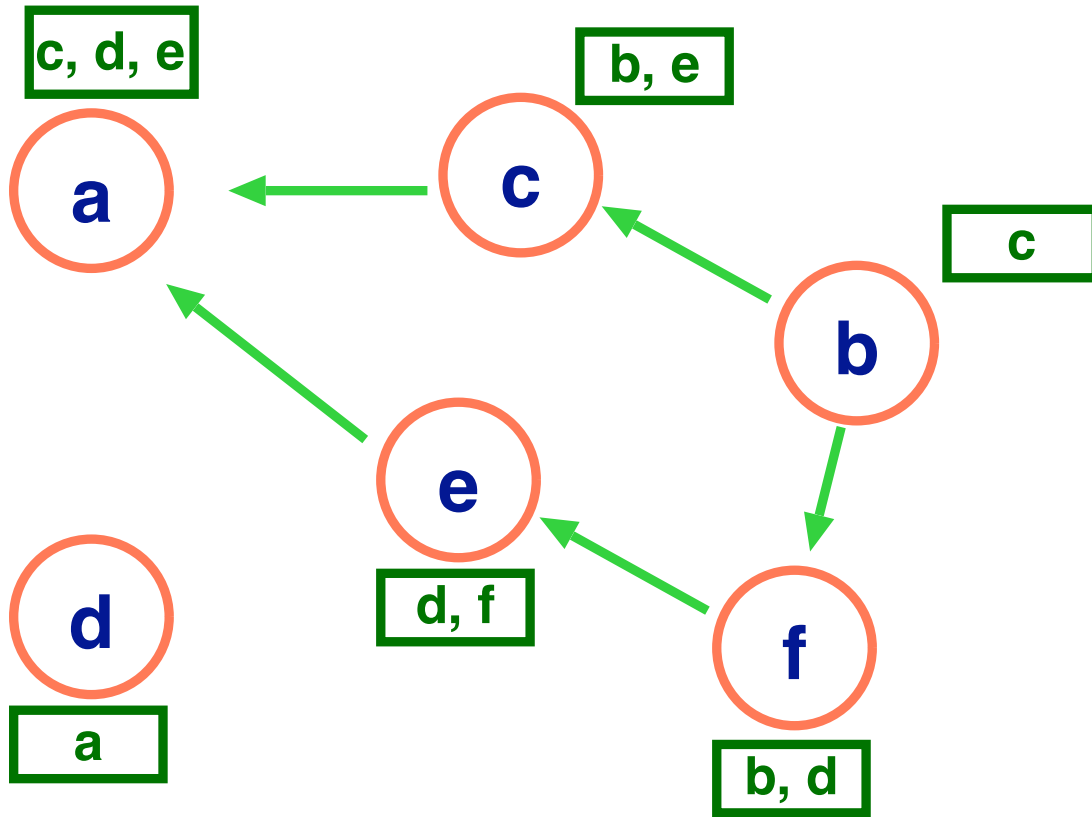
Flooding Example



Flooding Example



Flooding Example



- The $b \rightarrow a$ component of the trust graph is reconstructed via flooding

Analysis

- Flooding is “guaranteed” to reconstruct relevant trust graph component
- Problems:
 - Inefficient
 - Bad nodes can erase information about failed transactions by simply deleting low valued cookies!

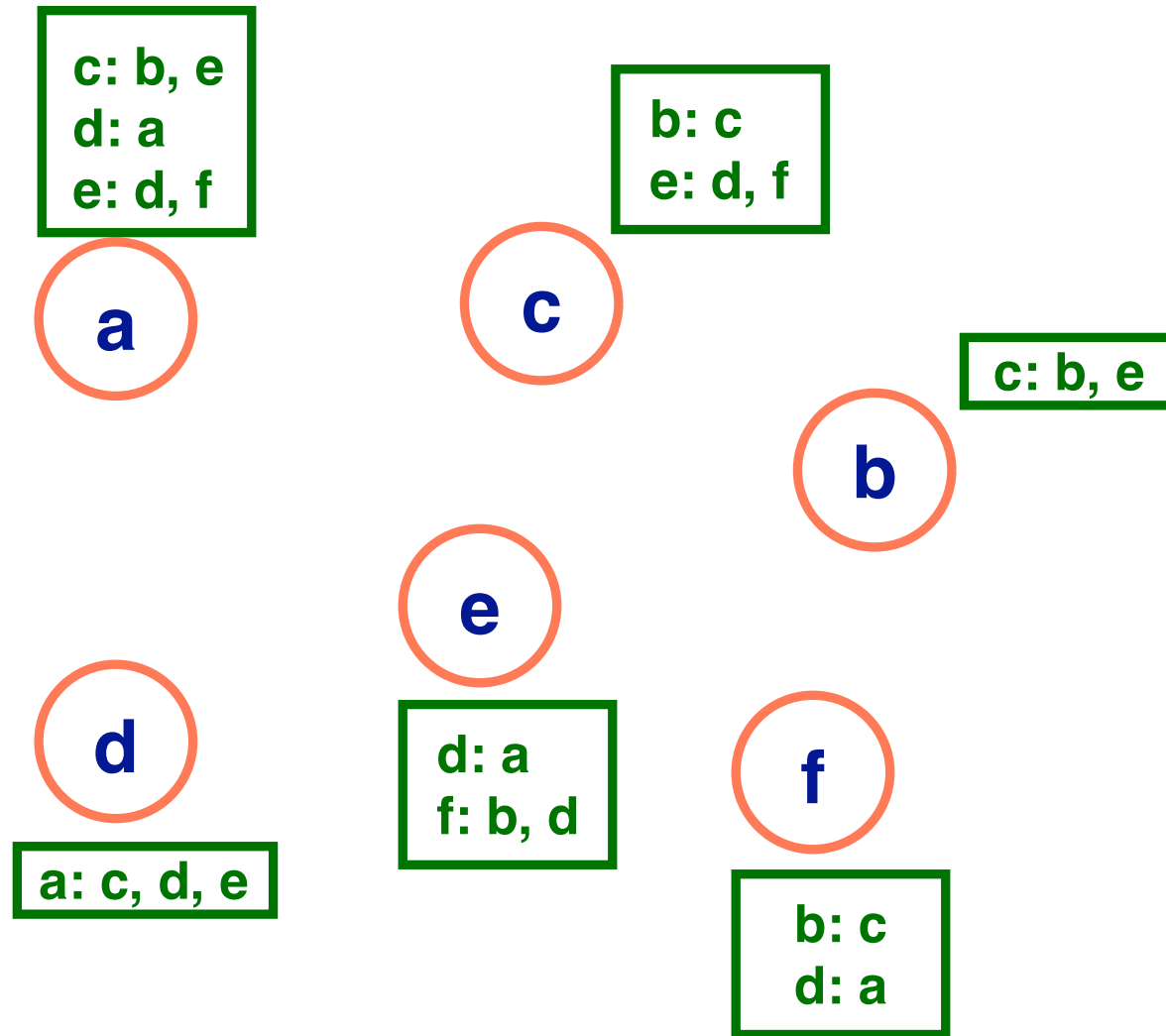
Refinements

- Search efficiency
 - Use cookie-digests to direct searches
 - Limit search outdegree
- Store failed transaction information in “negative cookies”
- Quickly discover good nodes using “preference lists”

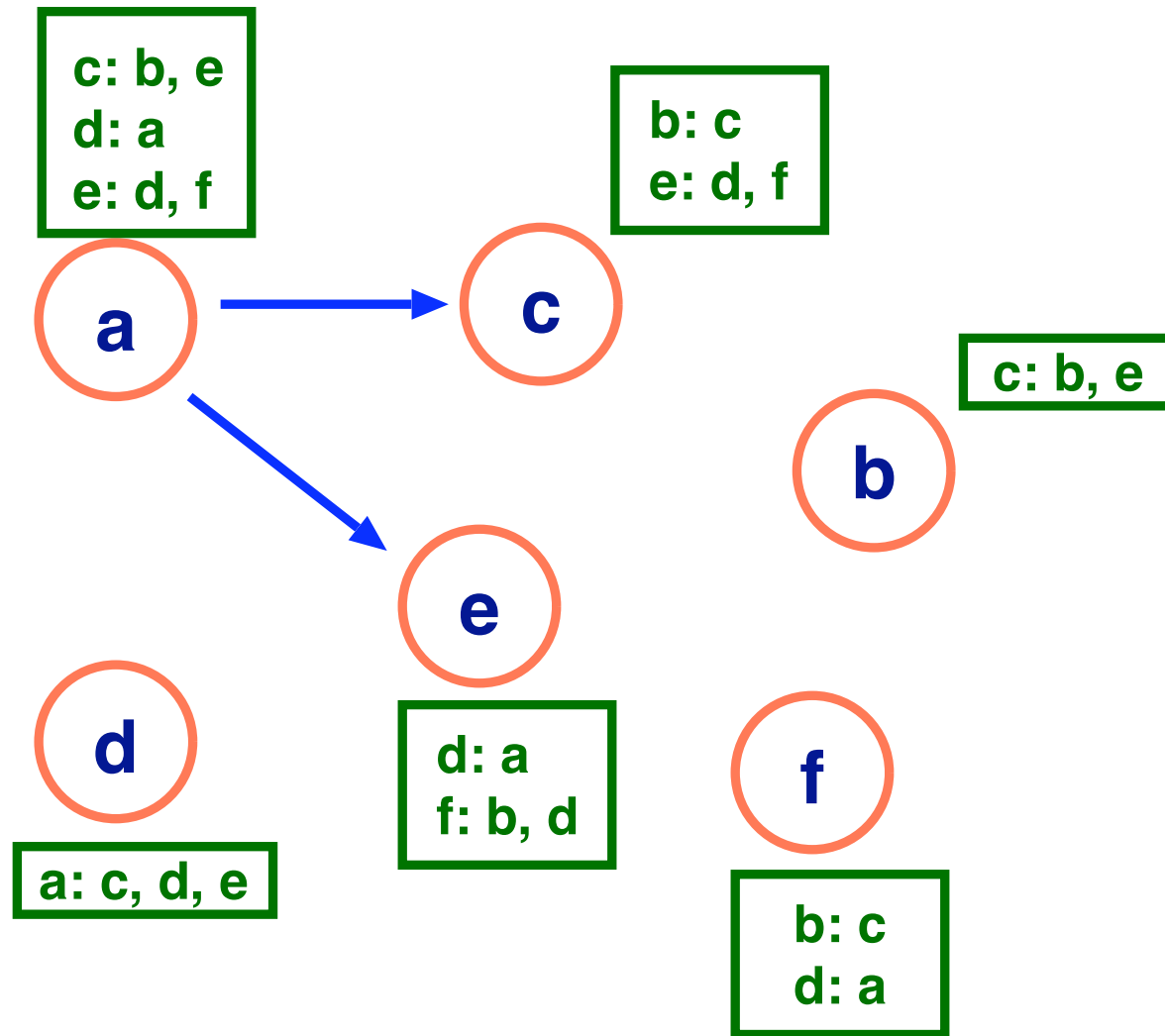
Using digests to speed up search

- Suppose Alice gives a cookie to Bob
- Along with the cookie, Alice also gives Bob a **digest** of all users from whom *she* has cookies
 - Cookie digests efficiently implemented using Bloom filters
- Searches proceed along random edges only for a hop or two
 - After a random search phase, searches are forwarded only if there is a hit in a cookie digest

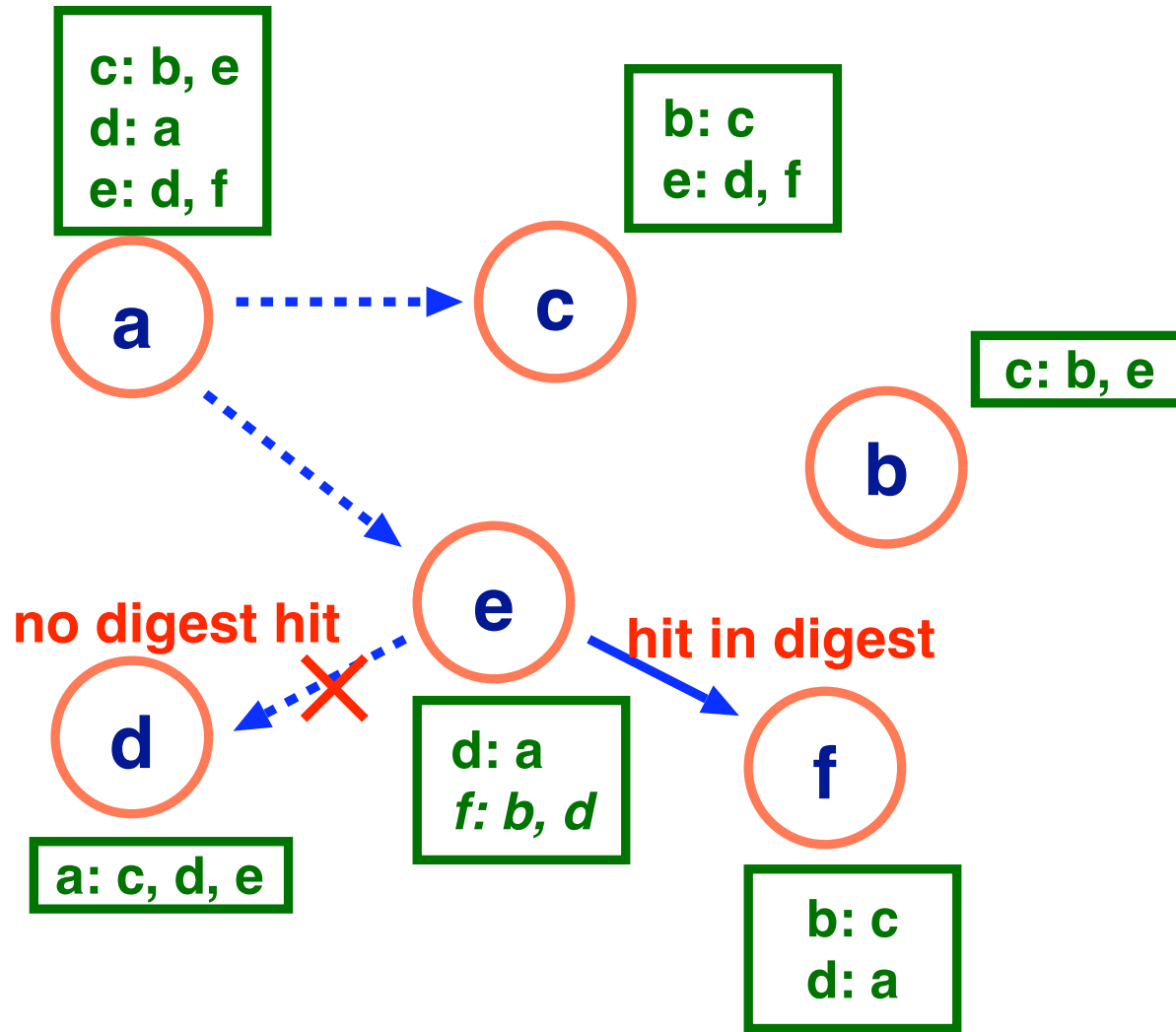
Digest-based search example



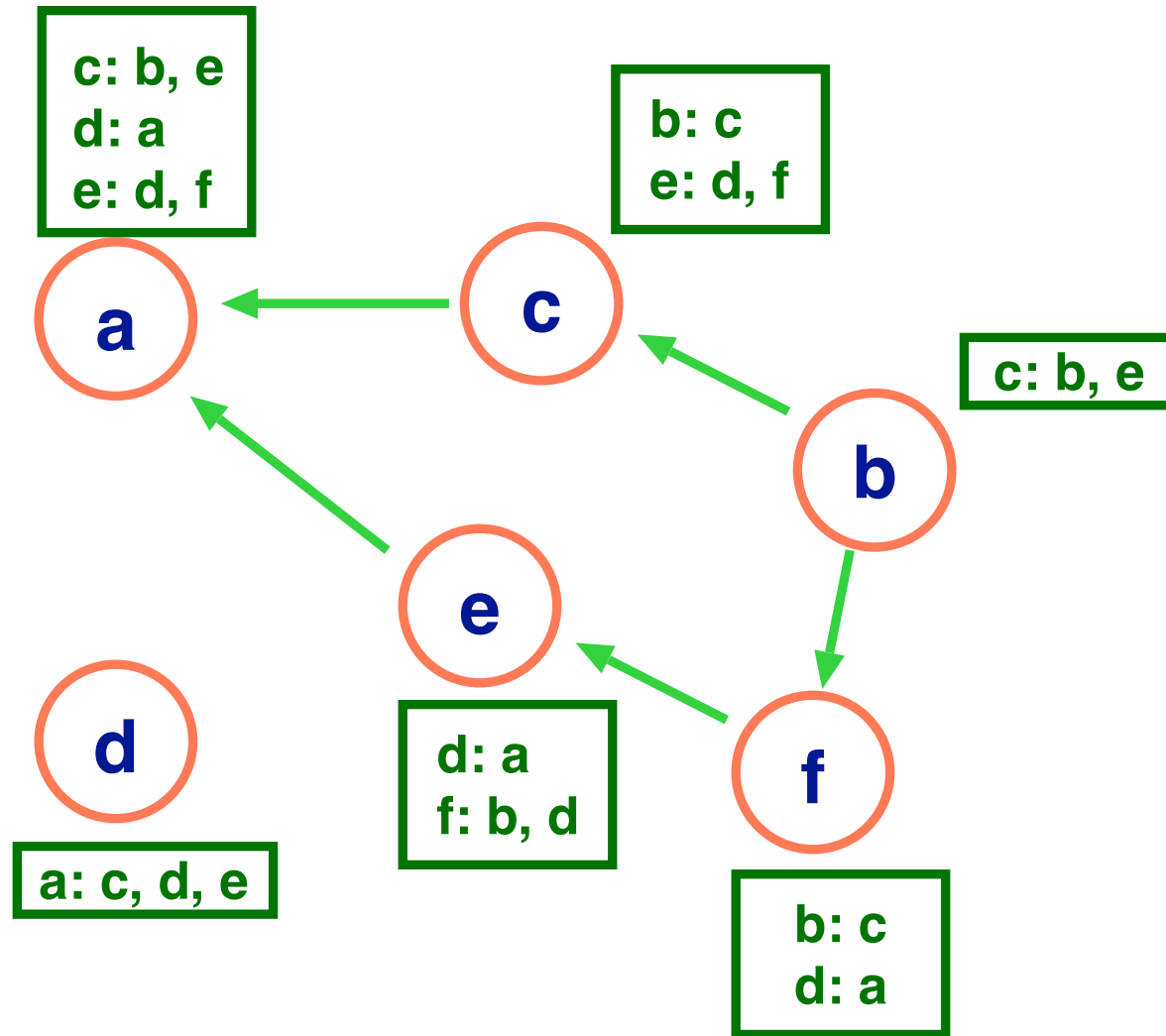
Digest-based search example



Digest-based search example



Digest-based search example



Negative Cookies

- Suppose Eve uses Alice's resources, but does not provide the negotiated resources she promised
- In the original scheme, Eve would receive a low-valued cookie from Alice. . . and promptly discard it
- Instead, Alice stores the low-valued "negative" cookie *herself*
 - Alice won't trust Eve as long as she stores the negative cookie
- The negative cookie can also be used by Bob (who trusts Alice)
 - Before accepting a transaction with Eve, Bob searches for negative cookies for Eve at users he trusts

Preference Lists

- To quickly discover other good nodes, each user (say Bob) keeps a preference list
- Bob's preference list contains potential high trust nodes (with whom Bob has not interacted yet)
- Bob interacts with nodes in the preference list with higher probability
- As Bob discovers new nodes during cookie searches, they are included in Bob's preference list iff they have high trust values

Thoughts

- All relies on past behavior
 - But this is similar to ebay/craig's list
- What if B doesn't agree w/ value of cookie A gave him?
 - Create negative cookie?
- Problematic in highly dynamic systems
- Not clear how trust inference would work in real world
 - A single path between A and B makes A susceptible to being taken.
 - Weighted approach seems to make most sense.