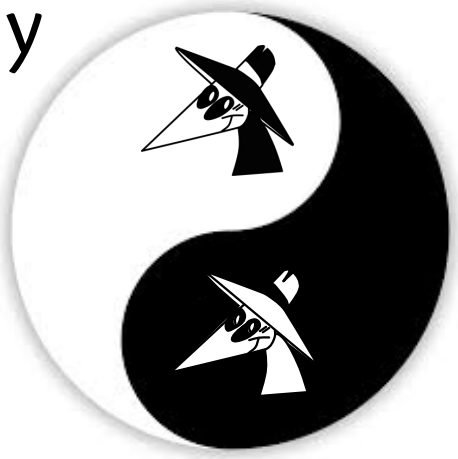
Toward compositional security and consistency

Andrew Myers

Cornell University

Programming with federation

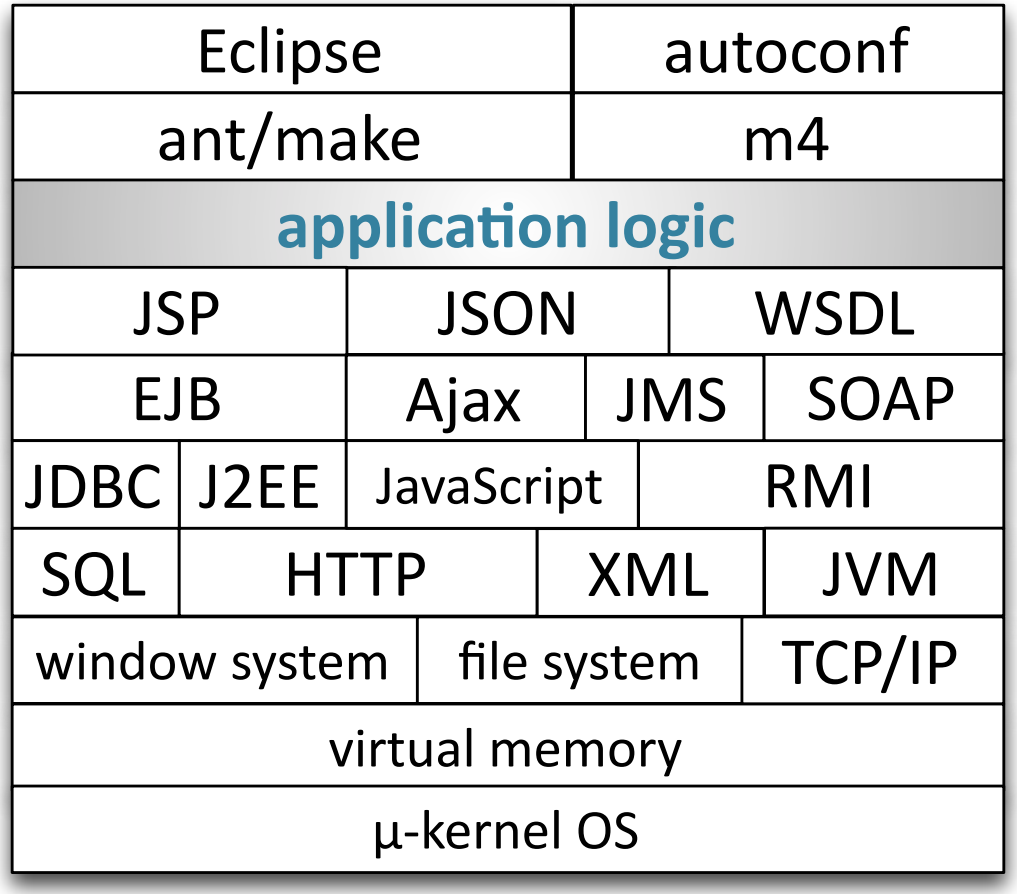
- Code, data, and computation increasingly composed across trust boundaries
  - users (browsers) vs. services
  - composing data: mashups
  - composing code: ads, libraries (jQuery, …)
  - composing services (authentication, paymt processing,…)
- Trust is complex and *heterogeneous*
  - different people trust code and data from different sources differently
- Applications worthy of trust by all participants?
  - protect users', providers' confidentiality & integrity, …



The API stack today

- More and more layers of languages and acronyms
  - leaky, fragile APIs
- Also need difficult protocols: cryptography, fault tolerance
- Complex, hard to reason about, and getting worse.
- Is there a better way?

A modern application



A higher-level abstraction?

Essential elements

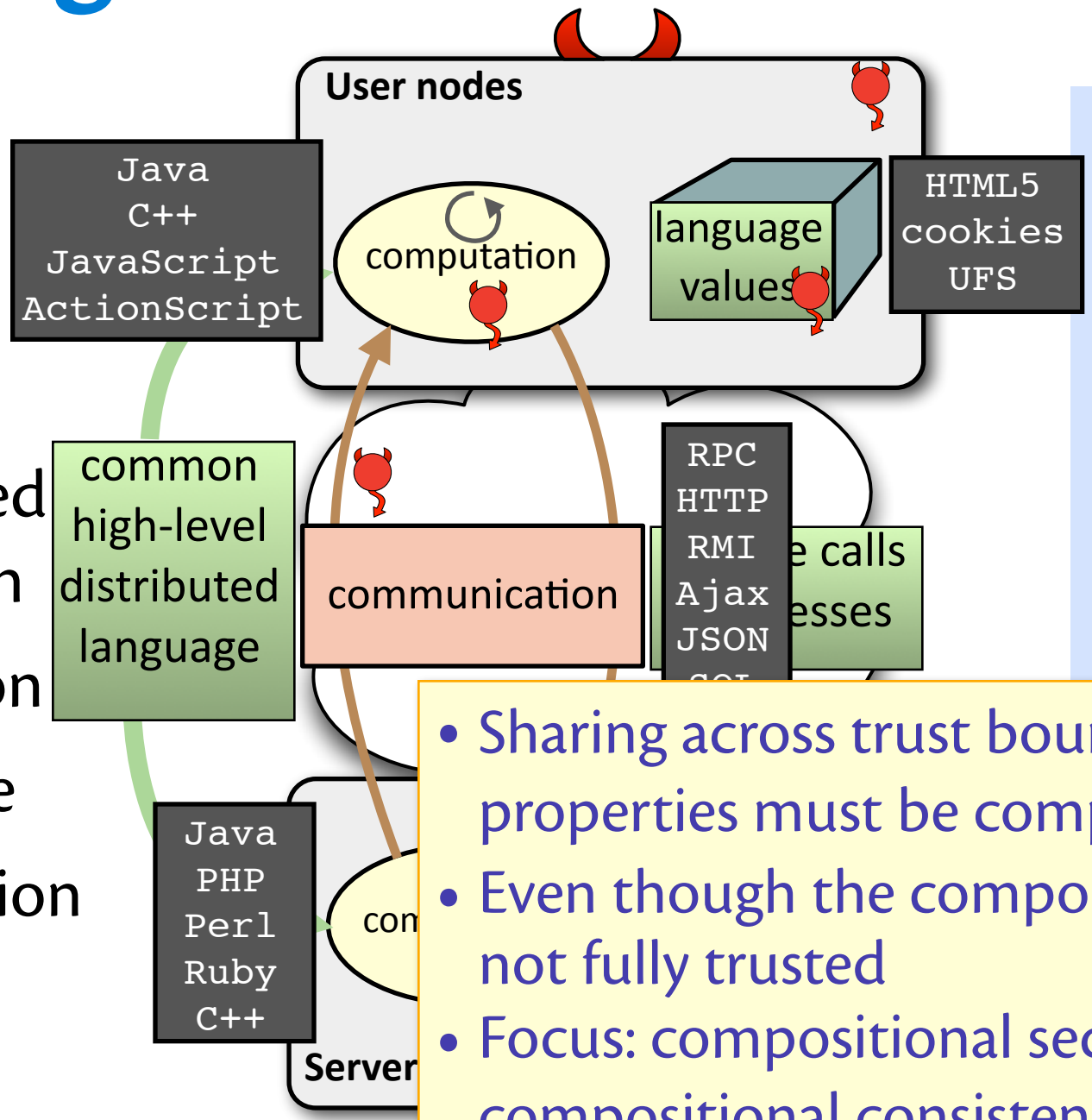
decentralized

distribution

computation

persistence

communication



Essential properties

security

consistency

durability

performance

availability

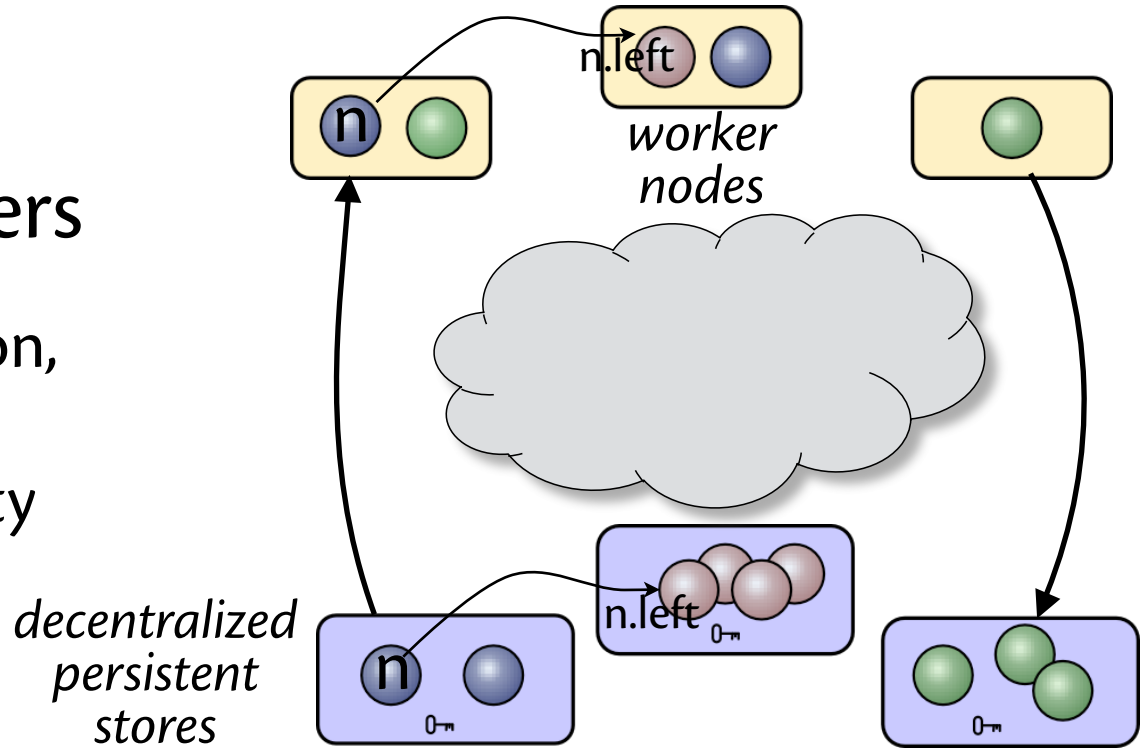
- Sharing across trust boundaries ⇒ properties must be compositional
- Even though the components are not fully trusted
- Focus: compositional security & compositional consistency

FABRIC: a language for secure distributed computing

- **Abstraction**:
  All information looks like an ordinary language-level object
- Java-like (Jif-like) language aimed at "CS200" programmers

. . . that supports distribution, persistent objects, strong (information-flow) security

n.left.value++



An untrusted mashup

Alice

3rd Party App:
"FriendMap"

Foursquare

Alice's friends

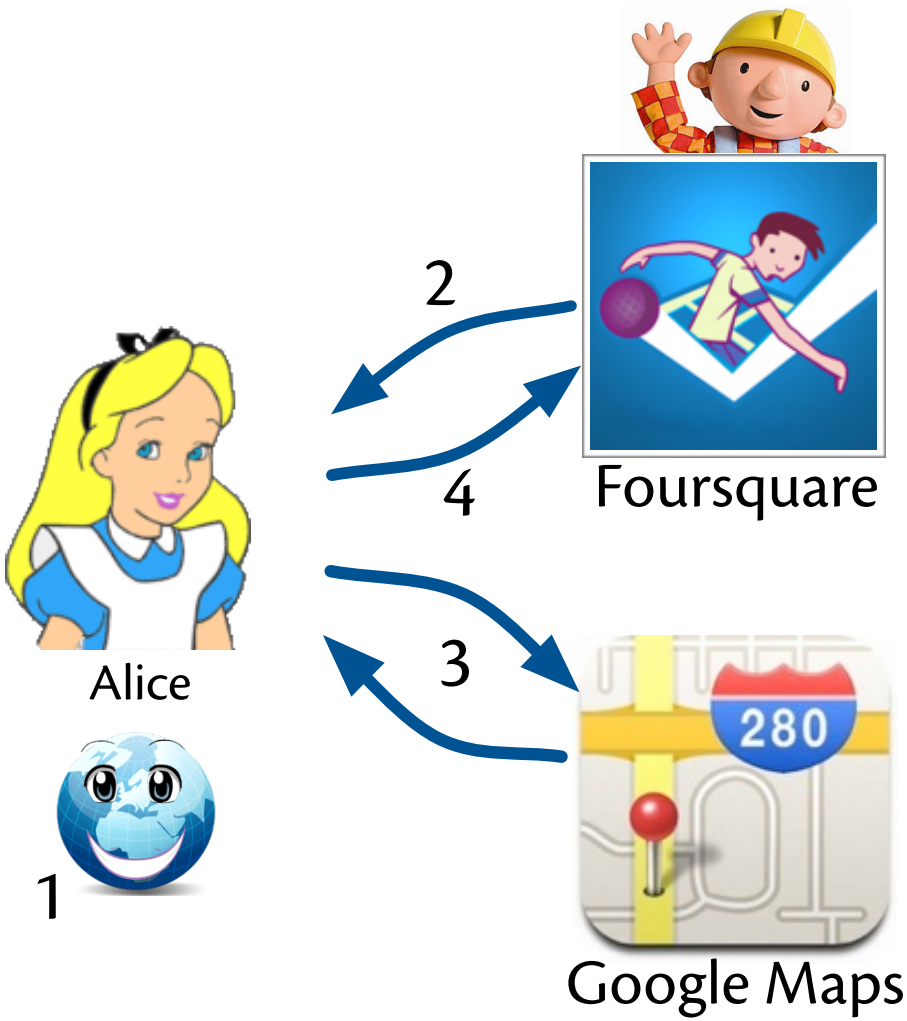
Bob

Google Maps

Not allowed now – violates same-origin policy!

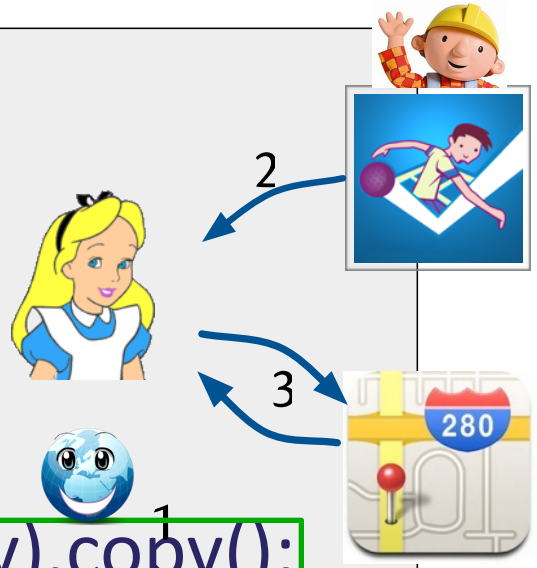
An untrusted mashup

1. Alice downloads and runs FriendMap
2. Fetch friends' locations
3. Get a map and place pins for nearby friends
4. Post to network



FriendMap: Simple code!

```
atomic {
  Box boundary = new Box();
  for (User friend : user.friends)
    boundary.expand(friend.location);
  Map map = ms.getMap@ms(boundary).copy();
  for (User friend : user.friends)
    map.addPin(annotated, friend.location, friend);
  return map;
}
```



Java plus:

- Transparent persistence
- Remote calls
- Atomic transactions
- Security annotations (elided)

- All objects accessed uniformly
- Remote calls are explicit (@)
- Code inside **atomic** executes atomically and in isolation.
- May include remote calls

Security in a world of distrust

- Open decentralized system—*anyone* can join
- **Decentralized security principle:**
  You can't be hurt by what you don't trust
  ⇒ Nodes you don't trust : malicious
- Explicit policies: need notion of "you" and "trust" in language

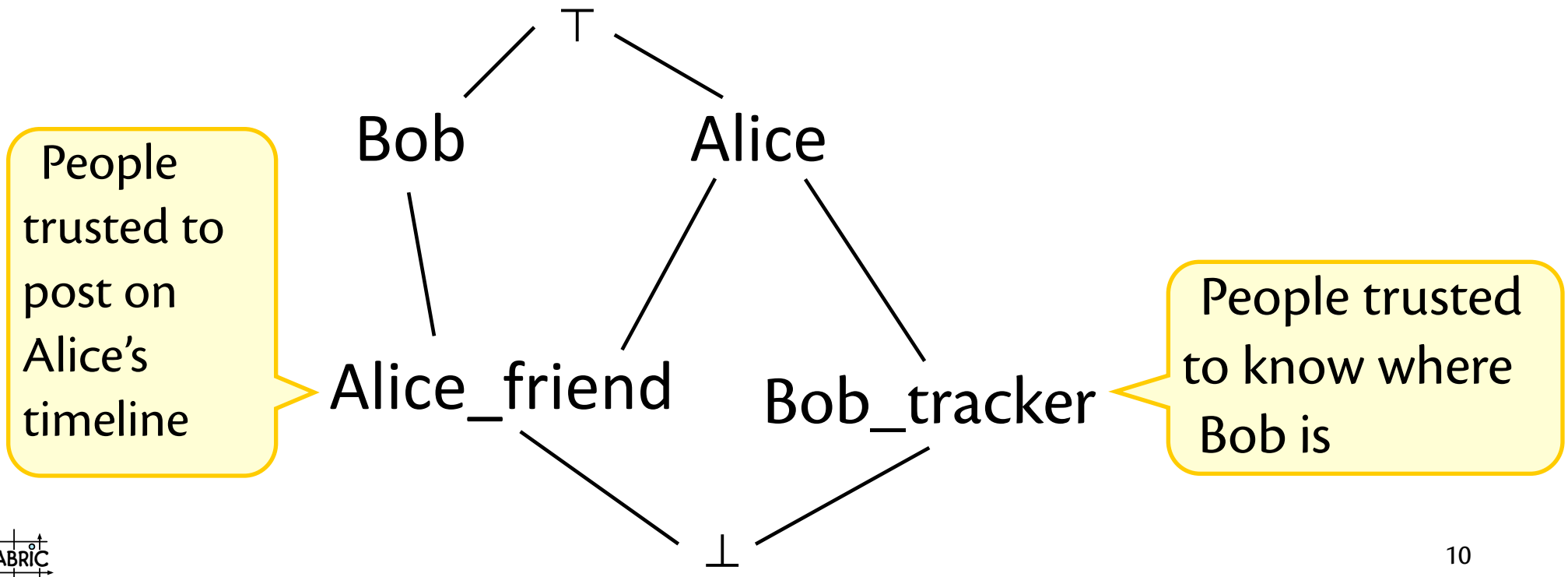
Principals and trust in Fabric

- **Principals** represent users, nodes, groups, roles, …
- Trust delegated via **acts-for (speaks-for)**

  "Alice acts-for Bob" (Alice ⩾ Bob) means "Bob trusts Alice"

  – Generates a **trust configuration**

  – Also: compound principals: Alice & Bob, Alice ∨ Bob



Security labels in Fabric

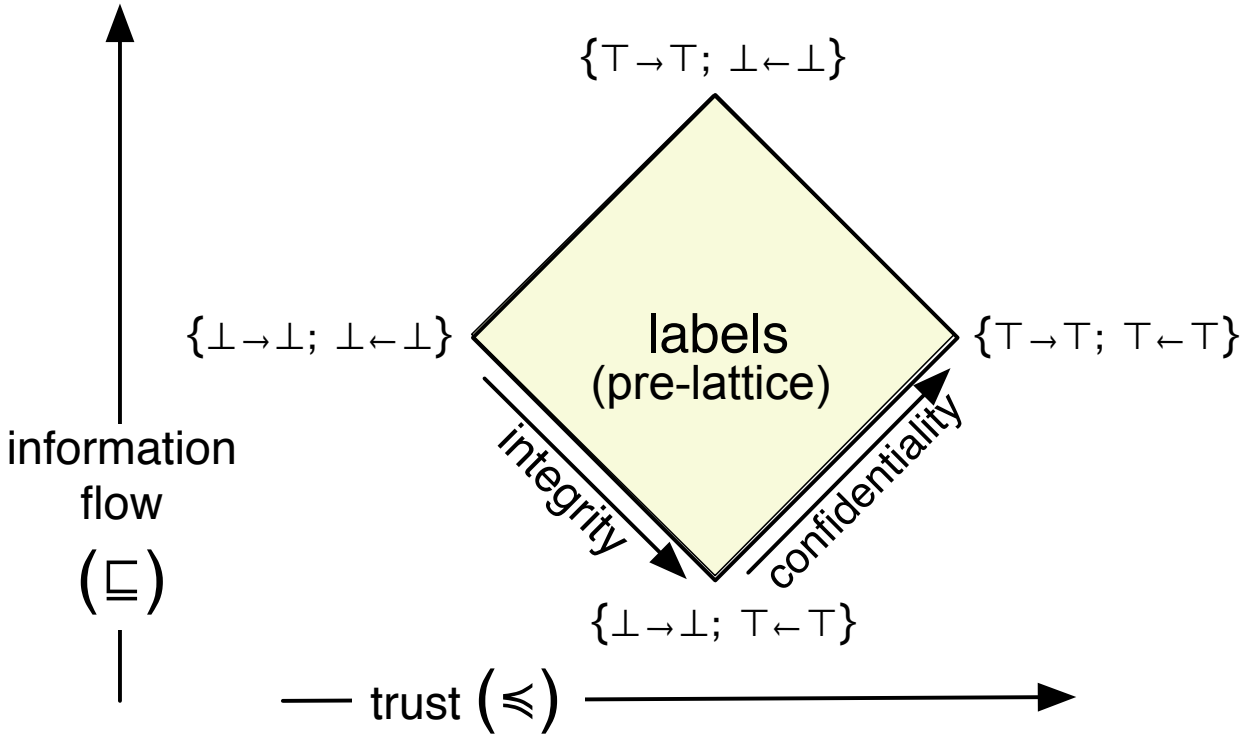
- Decentralized labels [Oakland'98] specify information policies

Confidentiality:	Alice	Alice permits Bob to *learn*
Integrity:	Alice ← Bob	Alice permits Bob to *affect*

```
class Timeline {
  String{Alice←Alice_friend; Alice→Alice_reader} text;
  …
}
```

Decentralized labels

- Base confidentiality policies $o \rightarrow r$, base integrity policies $o \leftarrow w$
- Labels are a product of confidentiality and integrity lattices
- Label ordering $L_1 \sqsubseteq L_2$ specifies allowed information flows
- Information flows can be checked statically:
  - for statement $x = y$, compiler checks $L_y \sqsubseteq L_x$



FriendMap, with labels

```
label fetchLbl = {resLbl ⊓ {⊤→ms}};
Box    boundary     = new Box();
for (User friend : user.friends)
   if (friendAccess ⊑ {⊤→friend.sn})          1
   &&  {friend→friend.locGrp} ⊑ fetchLbl)
      boundary.expand(friend.location);        2
Map map = ms.getMap@ms(boundary).copy(resLbl);
   for (User friend : user.friends)
      if (friendAccess ⊑ {⊤→friend.sn}          1
      &&  {friend→friend.locGrp} ⊑ resLbl)
         addPin(annotated, friend.location, friend);   3
   return map;
```

user

friend

friend.sn

ms

- To type-check, *dynamic* checks must be added to ensure:
  1. querying social network doesn't leak user's or friend's information
  2. friend permits map service to learn location
  3. friend permits location to be in posted result

Powers of the adversary

Adversary = any node not trusted to enforce security policy

- Strong adversary ⇒ strong security:
  - Observe, affect, remember any computation on its own nodes
  - Affect *any* object up to its level of integrity, in any Fabric node
  - See contents of *any* object up to its level of confidentiality
  - Supply malicious mobile code
  - Try to compromise and learn from distributed protocols
- We do ignore some side channels:
  - Timing messages over the network (but see [CCS'10, CCS'11, PLDI'12])
  - Network traffic analysis (Fabric does encrypt with SSL)

Semantic security

- Configurations s_1, s_2 indistinguishable to adversary at level A if $s_1 \approx_A s_2$.
  - Confidentiality: $\not\approx_A \Rightarrow$ adversary can *learn from* difference
  - Integrity: $\not\approx_A \Rightarrow$ adversary can *induce* difference
- Given system configuration s (code+data), denote traces of s as $[\![s]\!]$, lift $\approx_A$ to traces:

Noninterference [GM82] (essentially):

$$s_1 \approx_A s_2 \Rightarrow [\![s_1]\!] \approx_A [\![s_2]\!]$$

A 2-safety property

Beyond noninterference

- Some systems need to release information
  - declassify(e, L_1 to L_2)
- Some systems need to allow untrusted agents to have influence
  - endorse(e, L_1 to L_2)
- A 'type cast' on security labels, but:
  - **declassify**: Integrity of e and pc must be above confidentiality of relaxed policies in e
    idea: prevent laundering attacks by adversary
  - **integrity:** Integrity of pc must be above integrity of endorsed policies in e
    idea: prevent manufacturing integrity

Robustness

- Write **s[a]** for system **s** composed with 'attack' **a** must be **A**-integrity & non-interferent at **A**
- Declassify ⇒ **s[a]** *may* release information:

$$s_1 \approx_A s_2 \not\Rightarrow [\![s_1]\!] \approx_A [\![s_2]\!]$$

Robustness (essentially):

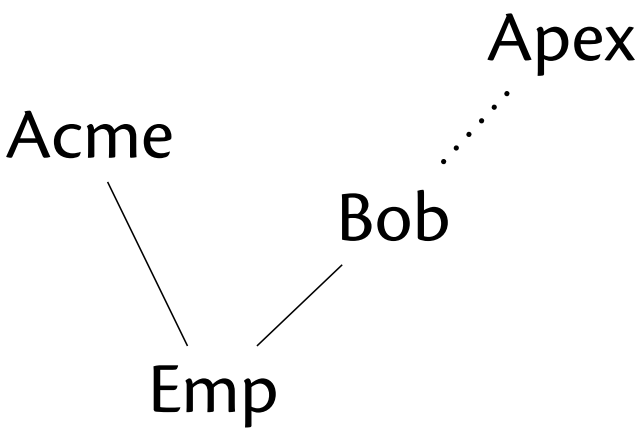
$$[\![s_1[a]]\!] \approx_A [\![s_2[a]]\!] \Rightarrow [\![s_1[a']]\!] \approx_A [\![s_2[a']]\!]$$

"All attacks equally effective"

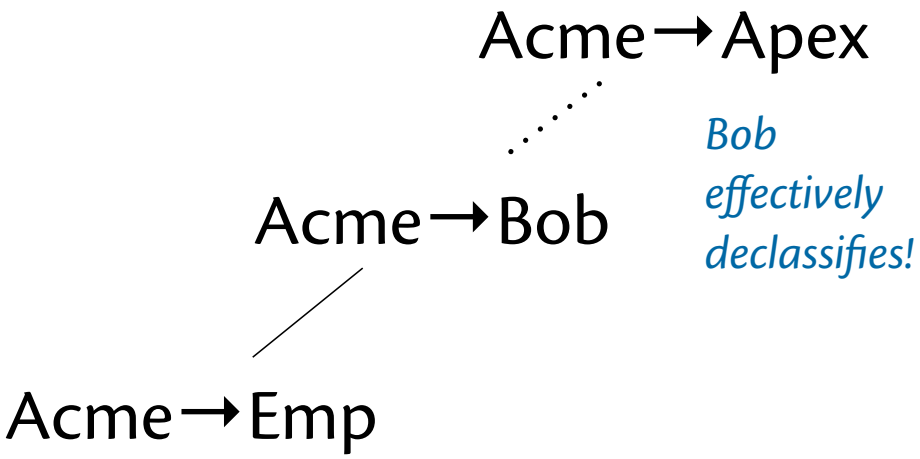
4 traces!

Dynamic trust

- Original label model assumed quasi-static *principal hierarchy*
- Fabric: Trust relationships p⩾q can appear/be revoked.



Trust ordering (⩾)



Information flow ordering (⊑)

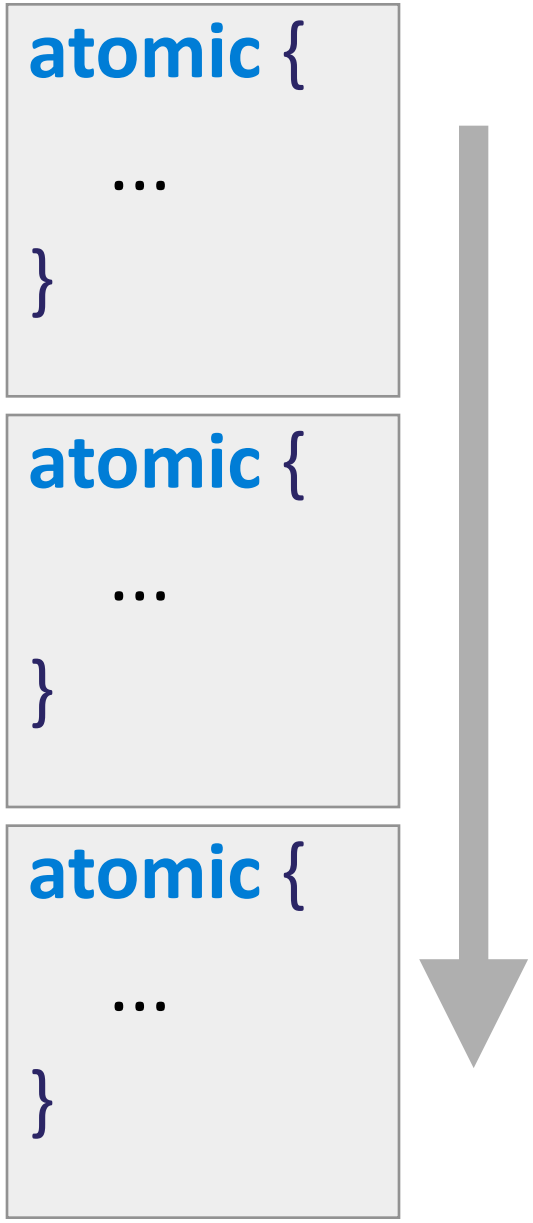
Robust delegation

- Delegation (authorization) interacts with information flow
  - delegation loophole
  - "poaching" (relabeling to escape later revocation)
  - information leakage via observation of delegation

    if (Bob ⩾ Alice) { … }
- Approach: **labeled trust configuration**
  - delegation relationships are information : may be confidential/ untrusted
  - relabeling $L_1 \sqsubseteq L_2$ must use high-integrity (relative to trust assumptions used) delegations
  - declassification & endorsement are special cases
  - can prove generalizations of NI/robustness.

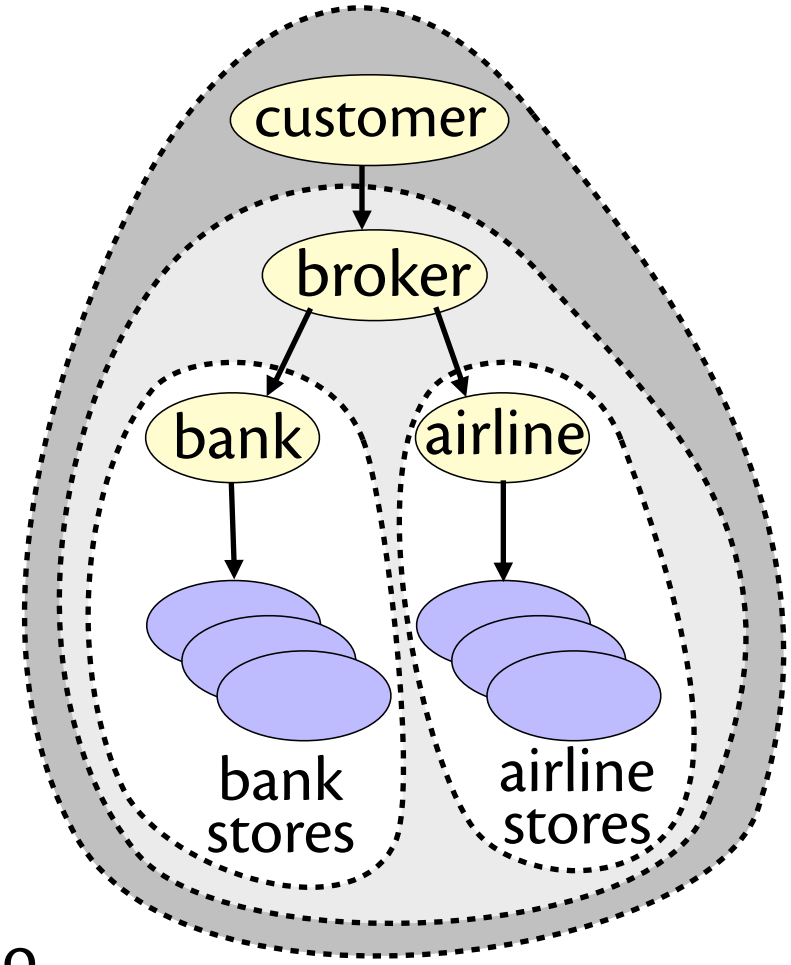
Challenge: consistency

- Weak/eventual consistency makes scaling easy, but programmers are bad at reasoning about it.
- Fabric offers *strict serializability*.
  - **atomic** block: atomicity, isolation from everything else, running on *any* node
  - Weak consistency on top of strong consistency? Easy (replicate, use small transactions)
  - Strong consistency on top of weak consistency? Hard, awkward
- Challenges: security and performance
- Implementation: optimistic, distributed transactions.



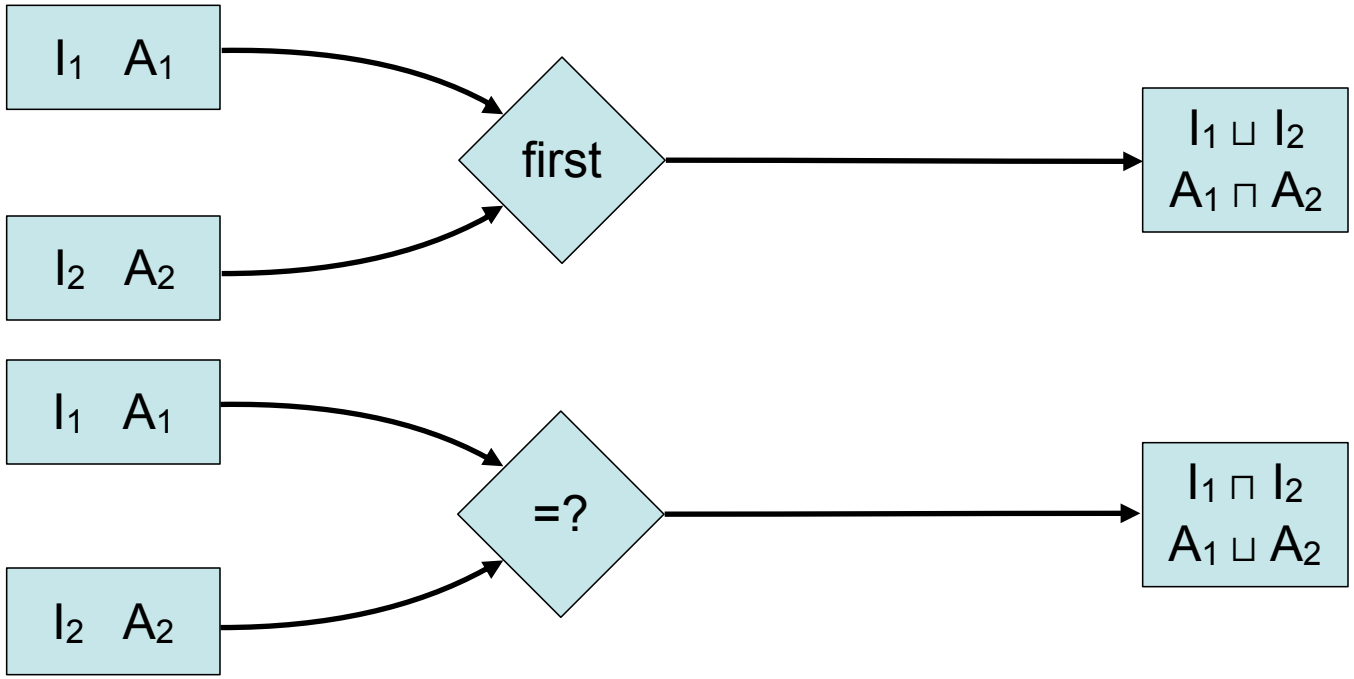
Secure hierarchical commit

- Security enforcement cannot depend on hosts not trusted to enforce policies.
- Untrusted workers in a transaction might:
  - not commit/update properly (violating integrity)
  - infer secrets from seeing transaction bookkeeping (violating confidentiality)
- Hierarchical 2PC ensures misbehaving nodes only break their own security
  - Example: buying a plane ticket. Trusted broker ensures full transaction commit so customer can't commit only half.
- Weakness: availability failures by coordinators treated as integrity failures



Availability vs. integrity

- View (eventual) availability as information flow label.
- Replication protocols create interaction:

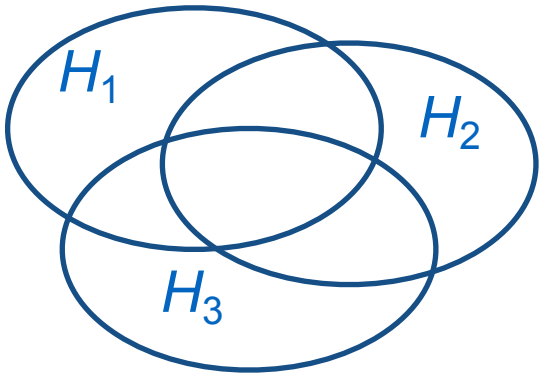


Boosting integrity & availability

High-integrity ~ F+1

Qualified ~ 2F+1

- Integrity of a set of (agreeing) hosts:
  $I(\{h_1,\ldots,h_n\}) = \sqcap\ I(h_i)$
- Result with integrity I is available if any qualified set *H* of hosts responds: (H cannot be separated into two low-integrity subsets)
  - If high-integrity subset *S* has same results, done
  - Also: prevents a low-integrity subset from lowering availability by responding with disagreeing results
  - $A = \sqcup_{\text{qualified subsets H}}\ A(H)$
- Generalizes various voting schemes
- Approach extends to other protocols:
  - e.g., fast consensus. (Paxos?)



Warranties

Warranty – a time-limited assertion about system state

- **State warranty** – about state of an object

  acct == {name: "Bob", bal: 42} until 2:00:02 p.m. (2 s)

- **Computation warranty** – about result of computation

  flight.seatsAvail(AISLE) >= 6 until 2:00:05 p.m. (5 s)

- Each warranty **defended** to ensure assertion remains true
- Duration set automatically, adaptively
  - Loosely synchronized clocks assumed (e.g., NTP)

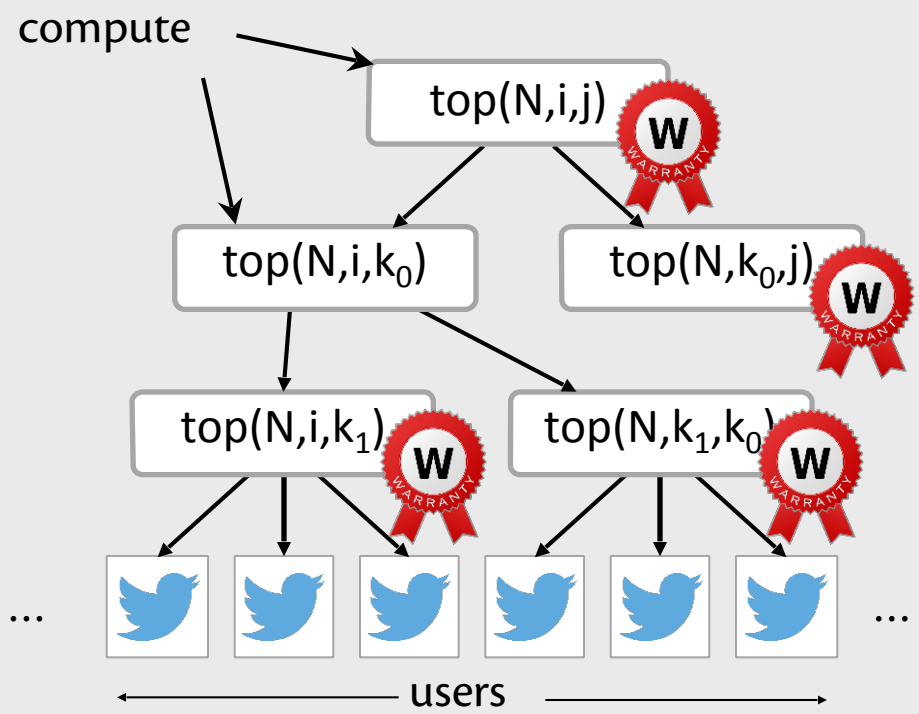
Avoid validating reads while guaranteeing **strict serializability** and **external consistency**

Computation warranties allow **caching computations with strong application-specific consistency**

Twitter analytics example

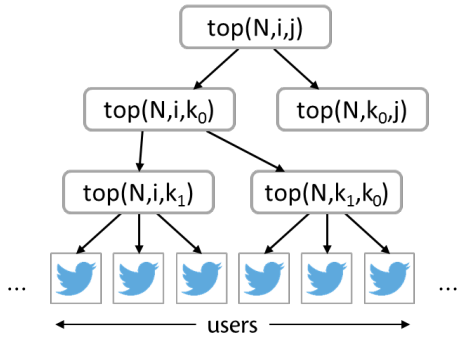
- Who are the top N most-followed Twitter users?
  - Unlikely to change often, though followers change frequently
- Warranty: n = top(N, i, j)
- Divide & conquer implementation
  - Allows incremental computation of new warranties

Warranty dependency tree



Evaluation: computation warranties

- "Twitter benchmark" : compute top 5 of 1000 users (e.g., home page)
  - divide & conquer
  - 98% reads (compute top 5)
  - 2% writes (follow/unfollow random user)

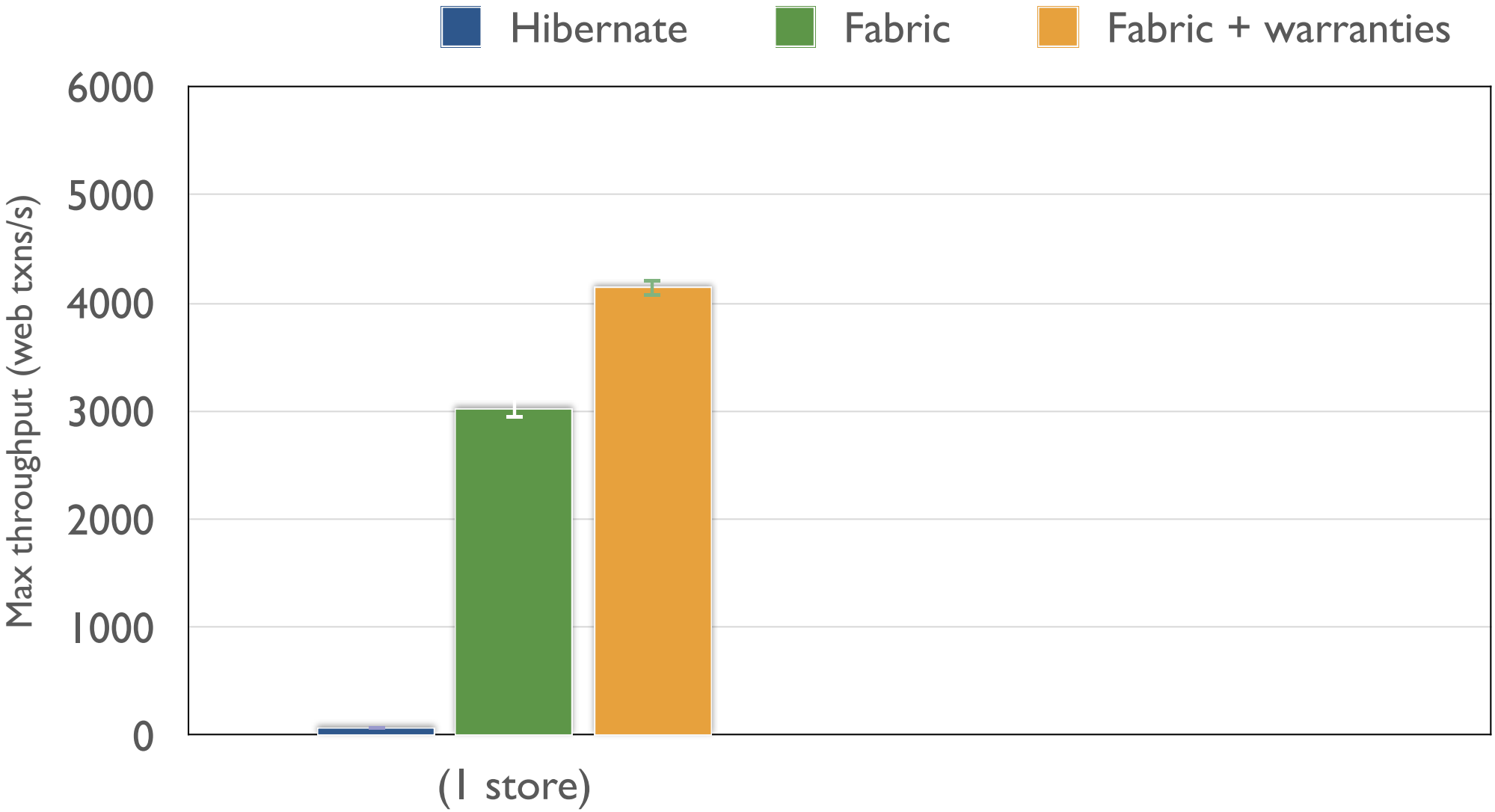


	Throughput (tx/s)	Median latency (ms)	95 write delay (ms)
Base Fabric	*17 ± 4*	*568 ± 354*	—
State warranties only	26 ± 5	1239 ± 455	623 ± 274
Comp. warranties	**343 ± 10**	**12 ± 2**	**16 ± 4**

Speedup via memoization, application-specific consistency

CMS throughput

1 store, 24 workers



- Fabric performance at least competitive with industry standard
- Language integration, collapsing API stack ⇒ more for less

Can we program the Internet Computer?

Users think programs run on the internet.

Programmers should think that way too.

Key: compositional abstractions