NSAC Network Security and Applied Cryptography Laboratory

http://crypto.cs.stonybrook.edu

The Blind Stone Tablet

Outsourcing Durability to Untrusted Parties

NDSS '09

Peter Williams
petertw@cs.stonybrook.edu

Radu Sion
sion@cs.stonybrook.edu

Dennis Shasha
shasha@cs.nyu.edu

STONY BROOK COMPUTER SCIENCE

NEW YORK UNIVERSITY

National Science Foundation
WHERE DISCOVERIES BEGIN

ver. 1 (02/4/2009)

Motivation for Outsourcing

- Hardware cheap, database reliability expensive.
- Redundant hardware, provision for disaster, specialized personnel.
- Let someone else to do it (“Provider”)

Problem with Outsourcing

- Provider may steal your secrets.
- Secrets can be worth billions.
- In some countries, a Provider employer is not even allowed to ask whether a prospective employee has been *convicted* of data theft.
- Contractual protections are mostly of the "best effort" kind, i.e. no protection at all.

What a Customer Wants

- Provider takes care of data durability.
- Clients enjoy a distributed database system with full transactional guarantees and full functionality (all of SQL or homegrown commands).
- Provider learns nothing!

Is Encryption Enough?

- Suppose we encrypt the data. Is that the end of the story?
- No, this makes searching expensive.
- No, because of various forms of traffic analysis.
- No, because server may violate serializability.

What do we want then?

- Access privacy: Provider cannot tell which data a client accesses.
- Full transaction semantics for distributed transactions.
- Good performance.

Can we get this?

Provider: Untrusted but Businesslike

- Provider is assumed to be curious (wants to know our data and is willing to do traffic analysis)
- Provider might try to put us in an inconsistent state.
- However, Provider does not want to be found out.

How about: outsource durability

- Client runs their own database but sends encrypted backups to the Provider

- But why stop there?

Outsource serialization as well

- Clients run local databases but synchronize via the Provider

Basic Strategy

- Each client holds a complete copy of the database (but may fail).
- Read-only transactions are completely local.
- Read-write (update/insert/delete) transactions are encrypted (using a private key shared by all clients) and pass through the Provider.
- All clients perform all transactions in same order.
- Provider holds log of encrypted transactions.

Algorithm 1:
global lock

- Client *c* does read-only transaction locally, without further ado.
- To do read-write transaction *t*, client *c* sends a request to Provider.
- Request is added to a queue.
- When all transactions previous to *t* have completed, *c* performs *t* locally and then sends updates that *t* performed to all other clients.

Algorithm 1: issues

- No concurrency.
- If *c* stops between the time it requests its slot and the time it performs *t*, no transaction following *t*'s slot can proceed.
- So, very sensitive to failure.

Algorithm 2: Precommit version

- Client *c* performs *t* locally on the state reflecting the first *k* committed transactions, but *c* does not commit *t*.
- Client *c* records updates *U* that *t* would have done.
- Client *c* sends *U* encrypted to Provider along with indication that *c* knows up to transaction *k*.

Algorithm 2: Precommit version continued

Provider sends to *c* all transactions that have committed or pre-committedd since transaction *k*

If any of those conflict with *t* then *c* aborts *t* else *c* commits *t*.

- Sites apply transactions that have committed.

Algorithm 2: issues

- More parallelism among non-conflicting transactions
- Could have livelock (repeated abort)
- If a transaction pre-commits but never commits, then a daemon process could see whether the transaction should abort or commit and do it (client sends up read set as well as updates)

Algorithm 2':
Optimistic version

- Client *c* performs *t* locally and then sends updates to Provider but does not roll back, still encrypted.
- Other steps the same.
- Probably better on the average.

Algorithm 3: motivation

- Algorithm 1 can be blocked if a single client fails.
- Algorithm 2 suffers from aborts, possible livelock, and the requirement of conflict detection.
- Is there an abort-free, detection-free, and wait-free alternative?

Algorithm 3:
abort-free, lock-free, wait-free

- In both algorithms 1 and 2, the client sends just the updates.
- Here the client sends the transaction text to the Provider, encrypted.
- The Provider simply sends this to all clients.
- All clients execute the transaction.

Text vs. updates

- Consider:

```
begin transaction
  x:= select max salary from emp
  if (x > 100000) then
    update sal = 1.1 * sal from emp
  else update sal = 1.2 * sal from emp
end transaction
```

Text vs. updates

- Text = whole transaction including conditional
- Updates = whichever update applies for current database state, e.g.
  update sal = 1.1 * sal from emp
  alone.

Algorithm 3: issues

- Requires transactions to be deterministic: depend on input parameters and state of database rather than on time of day, other timing, or random number.
- If transactions are non-deterministic, then transaction text could have different effects on different clients.
- For non-deterministic transactions, use algorithm 2.

General Issues

- How do we do failure recovery?
- How do we guarantee that Provider orders all transaction in the same way?

Failure Recovery

- Replay the log of all committed transactions. Could be very long.
- Clients periodically dump their database state up to a certain transaction number. Analogous to storing blood before going on a safari.

How Might Provider Sabotage Clients?

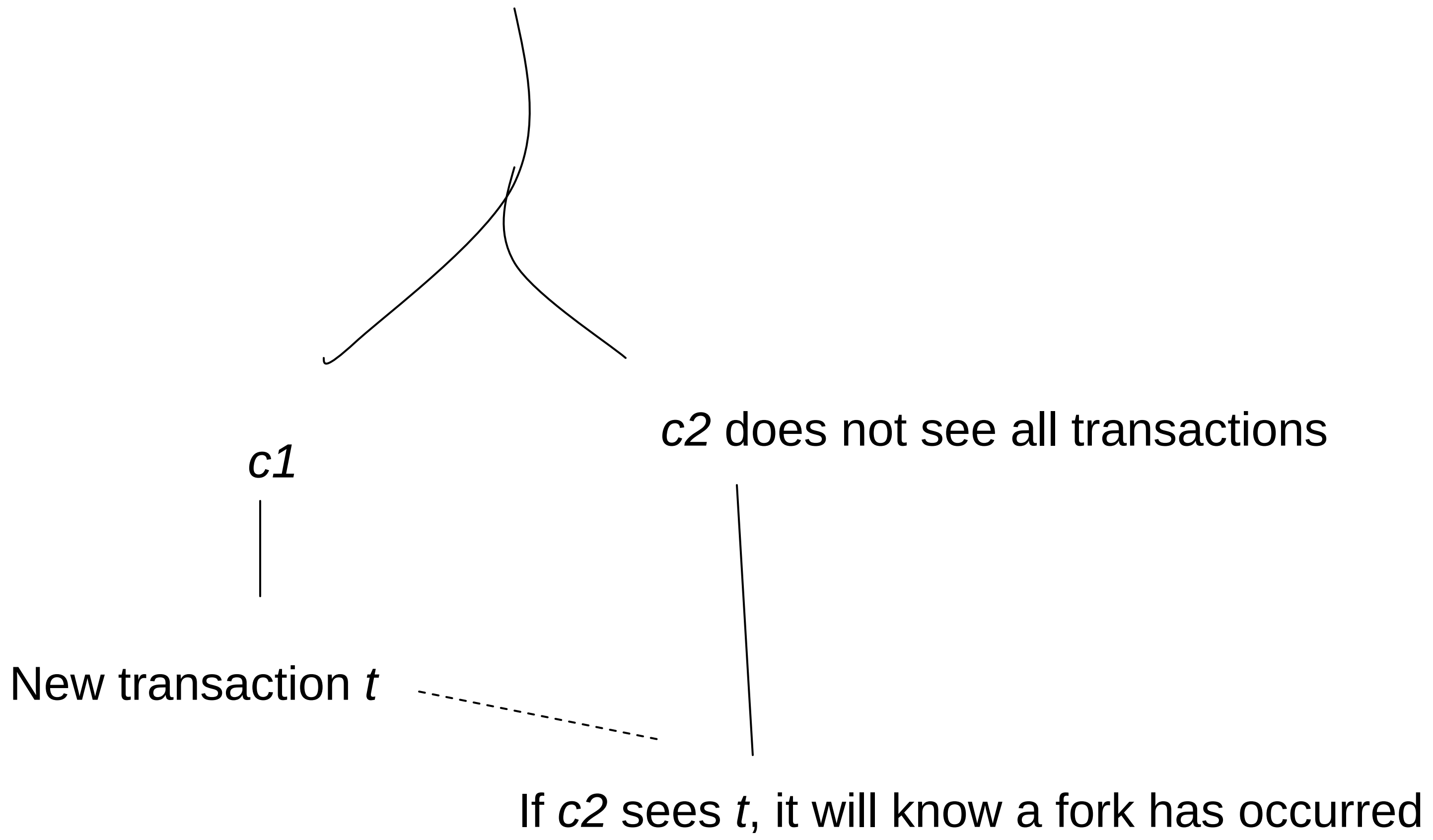
- Suppose that client *c1* performs transactions *t1* and *c2* performs *t2*.
- Untrusted server may show *t1* but not *t2* to some clients and *t1* but not *t2* to others and *t1* and *t2* to yet others.
- Would like to guarantee this can't happen.

Strategy to prevent sabotage I: fork consistency

- Fork consistency: if the Provider sends *c1* a transaction *t1* and then *t2* to *c1* but sends *t2* to *c2* without sending *t1* first, then if *c1* and *c2* exchange history data, Provider will be found out.

Fork Consistency in Pictures

- *c1* and *c2* forked



How to Encode Transaction History

- One way hash function H shared among clients.
- Hash chain of transaction encodings

  h0 = H(empty),

  h1 = H(h0, *t1*)

  h2 = H(h1, *t2*)

  …

How to Use Transaction History

- All clients when committing a new transaction *t* verify that their transaction history is the same as the history of the initiating client. If not, they know sabotage has occurred.

Strategy to prevent sabotage II: out-of-band communication

- Out-of-band communication: if *c1* and *c2* communicate an encoding of their transaction histories, they will know a sabotage has occurred.
- Net effect: Provider (businesslike) won't try this.

Summary

- A client company can contract with a Provider in full assurance that Provider cannot look at data or know which data is accessed.
- If Provider forks clients or denies service, it will be found out.
- Client can do all database operations.