Integrity and Consistency for Untrusted Services

Christian Cachin
IBM Research - Zurich

Joint work with Alex Shraer, Idit Keidar and many others.

22 June 2011
Where is my data?

1986  2011
Who runs my computation?

1986  2011
Remote storage - secure?

- Red Hat's servers were corrupted in Aug. '08
  - Package-signing key potentially exposed
  - Was source or binary content modified?
  - Red Hat stated in RHSA-2008:0855-6:
    ... we remain highly confident that our systems and processes prevented the intrusion from compromising RHN or the content distributed via RHN and accordingly believe that customers who keep their systems updated using Red Hat Network are not at risk.
Remote storage - correct?

- Amazon S3 silently corrupted data (20-Jun-08)
  - Cause was later traced to a defective router
Remote services - secure?

• City of Los Angeles moves 30,000 employees from desktops to the cloud (Google Apps)

• But:

  The Los Angeles police department (LAPD) is not ready to embrace the migration due to concerns with lags in e-mail delivery, and continued anxiety over the security of data entrusted to Google.

  “Google Apps Project Delays Highlight Cloud Security Concerns” [PCWorld, 26 Jul. 2010]

• Google installs a dedicated “Government Cloud” for U.S. federal/state/local customers
System model

- Server S
  - Normally correct
  - Sometimes faulty
    (untrusted, Byzantine)

- Clients: C₁ ... Cₙ
  - Correct, may crash
  - Run operations on server
  - Disconnected
  - Small trusted memory

- Asynchronous
Security issues

• Confidentiality
  - Expose stored data
  - Leak program code
  - Privacy of personally identifiable data

  → No concern here

• Integrity
  - Modify stored data
  - Computation returns incorrect results
  - Responses to clients not consistent

  → Focus of this work
Part 1: Storage
Storage model

- Functionality **MEM**
  - Array of registers $x_1 \ldots x_n$
  - Two operations
    - read($i$) $\rightarrow x_i$ returns $x_i$
    - write($i,x$) $\rightarrow$ ok updates $x_i$ to new value $x$

- Operations should be atomic

- Abstraction of shared memory

- Previous work on forking consistency conditions only considered **MEM**
  [MS02] [LKMS04] [CSS07] [CKS09] ...
Untrusted storage

• Clients interact with service through operations to read/write data

• Clients may digitally sign their write requests
  → Server cannot forge read values
  → But answer with outdated values ("replay attack")
  → But send different values to different clients (violates consistency)
The problem

C_1 \rightarrow \text{write}(1,x) \rightarrow \text{write}(1,u) \rightarrow \text{write}(1,t) \rightarrow \text{C}_3

C_1 \rightarrow \text{write}(2,v) \rightarrow \text{read}(1) \rightarrow x \rightarrow \text{write}(2,w) \rightarrow \text{C}_3

C_2 \rightarrow \text{read}(1) \rightarrow u \rightarrow \text{read}(2) \rightarrow w
Solution: Fork-linearizability

- Server may present different views to clients
  - “Fork” their views of history
  - Clients cannot prevent this

- Fork linearizability [MS02]
  - If server forks the views of two clients once, then
    → their views are forked ever after
    → they never again see each others updates

- Every inconsistency results in a fork
  - Not possible to cover up

- Forks can be detected on separate channel
  - Best achievable guarantee with faulty server
Fork-linearizability graphically

C₁  write(1,x)  write(1,u)  write(1,t)  C₂  write(2,v)  read(1) → x  write(2,w)  C₃  read(1) → u  read(2) → w

View of C₁
View of C₂
View of C₃
**Background: Semantics of concurrent operations**

**Safe:** Every read not concurrent with a write returns the most recently written value.

*(Regular: Safe & any read concurrent with a write returns either the most recently written value or the concurrently written value: C₃ may read x or u.)*

**Atomic:** Regular & all read and write operations are linearizable: C₃ must read u.
Every operation appears to execute atomically at its linearization point, which lies in real time between invocation and response.
Linearizability formally

A history $\sigma$ is linearizable (w.r.t. $F$)

$\Leftrightarrow \exists$ permutation $\pi$ of $\sigma$ such that:
- $\pi$ is sequential and follows specification (of $F$);
- $\forall i$ all operations of $C_i$ are in $\sigma$;
- $\pi$ preserves real-time order of $\sigma$. 
A history $\sigma$ is fork-linearizable

$\leftrightarrow \forall i \exists$ subset $\sigma_i \subseteq \sigma$ and permutation $\pi_i$ of $\sigma_i$ s.t.

- All operations of $C_i$ are in $\sigma_i$;
- $\pi_i$ is sequential and follows specification;
- If $o \in \pi_i \cap \pi_j$, then $\pi_i = \pi_j$ up to $o$;
- $\pi_i$ preserves real-time order of $\sigma_i$. 
Fork-linearizable Byzantine emulations

- Protocol $P$ emulates functionality $F$ on a Byzantine server $S$ with fork-linearizability, whenever
  - If $S$ correct, then history of every (...) execution of $P$ is linearizable w.r.t. $F$;
  - The history of every (...) execution of $P$ is fork-linearizable w.r.t. $F$.

[CSS07]
A trivial protocol

- Fork-linearizable Byzantine emulation

- Idea [MS02]: sign the complete history
  - Server sends history with all signatures
  - Client verifies all operations and signatures
  - Client adds its operation and signs new history

- Impractical since messages and history grow with system age
Fork-linearizable storage (1)

- **Client** $C_i$
  - Stores timestamp $t_i$ and
  - Version (vector of timestamps) $T$, where $T[i] = t_i$
  - Increments $t_i$ and updates $T$ at every operation

- **Versions order operations**
  - After every operation, client signs new timestamp, version, and data
    $$V = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

- **Verification with version** $T$ of last operation
  - Version $V$ of next operation must be $V \geq T$
  - Signatures must verify
Fork-linearizable storage (2)

Version $T = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$

$V \geq T$ ?
verify $V[i] = T[i]$ ?
verify($\sigma, V|...x_j$) ?
if not then abort
$T := V$; $T[i] := T[i]+1$
$\phi := \text{sign}(T|...)$
return $x_j$

Version $V = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$
Memory $x_1 \ldots x_n$
Signature $\sigma$ from $C_c$

$\text{SUBMIT, READ, j}$

$\text{REPLY, V, ...x_j, c, } \sigma$

$\text{COMMIT, T, } \phi$

$V := T$
$\sigma := \phi$
$c := i$
Fork-linearizable storage (3)

- If clients are forked, they sign and store incomparable versions
  \[
  \begin{array}{ccc}
  u \\
  v \\
  w + 1 \\
  \end{array}
  \quad ?
  \quad
  \begin{array}{ccc}
  u \\
  v + 1 \\
  w \\
  \end{array}
  \]

- Signatures prevent server from other manipulations
  - Protocol uses $O(n)$ memory for emulating fork-linearizable shared memory on Byzantine server

- Increasing concurrency?
  - Here, clients proceed in lock-step mode
  - Yes, but see papers...
**Fork-linearizability benefits**

- Client $C_i$ writes many values $u, v, w, x \ldots$

- Without protection, faulty $S$ may return any of those values to a reader $C_i$

- With fork-linearizable emulation
  - $C_i$ writes $z$ and tells $C_j$ out-of-band
  - $C_i$ reads $r$ from location $i$
    - if $r = z$, then all values read so far were correct
    - if $r \neq z$, then $S$ is faulty
  - Out-of-band might be only synchronized clocks
Storage systems providing fork-linearizability

SUNDR [LKMS04] Secure untrusted data repository
- NFS network file system API
- Extensions to NFS server and NFS client
- Hash tree over all files owned by every user

CSVN [CG09] Integrity-protecting Subversion revision-control system
- SVN operations are verified
- Hash tree over file repository
- Based on fork-linearizable storage protocol [CSS07]
Hash trees for integrity checking (Merkle)

- Parent node is hash of its children
- Root hash value commits all data blocks
  - Root hash in trusted memory
  - Tree is on extra untrusted storage
- To verify $x_i$, recompute path from $x_i$ to root with sibling nodes and compare to trusted root hash
- To update $x_i$, recompute new root and nodes along path from $x_i$ to root

Read & write operations in $O(\log n)$ work
- hash operations
- extra storage accesses
Forking consistency is blocking

- All fork-linearizable emulations of storage have executions with a correct $S$, where $C_i$ must wait for $C_j$ [CSS07].

- Also fork-sequentially consistent storage emulation protocols [OR06] are blocking.

- Also fork-*-linearizable storage emulation protocols [LM07] are blocking.

$\rightarrow$ Weak forklinearizability allows wait-free emulations [CKS09]
Part 2: General services
From storage to any service

- Server provides any deterministic functionality $F$

- $F$ is a state machine: $F(s,o) \rightarrow (s',r)$
  - State $s$
  - Operation $o$
  - New state $s'$
  - Result $r$

- Previous work on forking consistency conditions only considers MEM function
  [MS02] [LKMS04] [CSS07] [CKS09] ...
Background: Authenticated data structures

- Server stores (large) state $s$ of $DB$ (read-only)
- Client has (short) trusted authenticator $a$

Syntax
- Client sends query $q$ to $S$
- $S$ sends result $r := DB(s,q)$
- Client runs algorithm $\text{verify}(a,q,r) \rightarrow \text{OK} / \text{FAIL}$

Correctness & security
- If $a$ is correct authenticator for $s$ and $DB(s,q)=r$ then $\text{verify}(a,q,r) \rightarrow \text{OK}$
- No faulty $S$ can forge $q^*$ and $r^* \neq DB(s,q^*)$ with $\text{verify}(a,q^*,r^*) \rightarrow \text{OK}$
Authenticated separated exec.

- Generalizes authenticated data structures
  - Client maintains authenticator \( a \) in trusted memory

- Execution separated between \( S \) and \( C_i \)

\[
\begin{align*}
\text{Server } S & \quad s \\
& \quad o \\
& \quad (s', r) = F(s, o) \\

\text{Client } C_i & \quad \text{a}' \\
& \quad |s_0| \ll |s| \\
\end{align*}
\]

\[
\begin{align*}
s_0 & := a\text{-extract}(s, o) \\
\text{(a', s_0', r) := a\text{-exec}(a, s_0, o)} \quad \text{if } r = \text{FAIL} \text{ then} \\
& \quad \text{output FAIL} \\
& \quad \text{else} \\
& \quad \text{output } r \\

s' & := a\text{-reconcile}(s, s_0', o) \\
\end{align*}
\]
Properties of authenticated separated execution

- **Correctness**
  - Result computed by separated execution of operation \( o \) under \( F \) on state \( s \) is equal to \( F(s,o) \)
  - Size of \( s_0 \) and \( s_0' \) is much smaller than size of \( s \)

- **Security**
  - Given proper authenticator \( a \) for state \( s \), then for any \( o^* \) and \( s^*_o \) (forged by faulty \( S \)):
    \[
    a-exec(a,s^*_o,o^*) \rightarrow \text{FAIL} \text{ or } a-exec(a,s^*_o,o^*) \rightarrow r \text{ with } r=F(s,o)
    \]
  - I.e., client detects modification or gets correct result
Fork-linearizable service impl.

Version $T$

$V \geq T$?

verify $V[i] = T[i]$?

verify($\sigma$, $V|a$)?

if not then abort

$(a', s_o', r) := a$-exec($a, s_o, o$)

if $r = \text{FAIL}$ then abort

$T := V$; $T[i] := T[i]+1$

$\phi := \text{sign}(T|a')$

[SUBMIT, $o$]

[REPLY, $V$, $a$, $s_o$, $c$, $\sigma$]

Version $V$

State $s$

Authenticator $a$

Signature $\sigma$ from $C_C$

$s_o := a$-extract($s, o$)

[s := a-reconcile($s, s_o', o$)

$V := T$

$a := a'$; $\sigma := \phi$; $c := i$}

[COMMIT, $T$, $a'$, $s_o'$, $\phi$]
Protocol properties

- If server correct, then linearizable
  - Correct server schedules operations as they arrive

- With faulty server, still fork-linearizable w.r.t. $F$
  - From properties of versions and authenticated execution scheme

- Complexity
  - Three messages
  - Message size $O(n)$, with $n$ clients
Insight

- New protocol extends all existing protocols for untrusted storage
- Generalizes untrusted remote execution protocols for other (limited) functions
- Relevant for applications run in "clouds"
Related work

[WSS09] Blind Stone Tablet
- Run relational database on untrusted server

[FZFF10] SPORC: Group collaboration on ...
- Shared editor for working on documents in cloud
- Operational transforms let operations commute

[MSLCADW10] Depot: Cloud storage
- Respects fork-causality consistency notion
Conclusion

- Remote checking for applications in cloud

- Target is collaboration among group of mutually trusting clients

- Fork-linearizable service execution protocol
  - In normal case, linearizable and “blocking”
  - In failure case, respects fork-linearizability

- Ongoing and future work on extending the protocol to avoid “blocking”
References


• C. Cachin, M. Geisler. Integrity Protection for Revision Control. ACNS 2009.


Introduction to Reliable and Secure Distributed Programming

- C. Cachin, R. Guerraoui, L. Rodrigues
- 2nd ed. of Introduction to Reliable Distributed Programming
- Springer, 2011

Web: www.distributedprogramming.net
Backup
Emulating fork-linearizable memory requires waiting

**Theorem:** Every protocol has executions with a correct server where a client $C_i$ must wait for some client $C_j$.

**Proof sketch:**
- Protocol with only 1 round of messages
- Assume such a protocol exists
- Construct an execution that is not fork-linearizable, but looks like one that is to every client
**Proof for 1-round protocols**

**Correct S:**

C1: \( w_1(u) \)

S: \( * \)

C2: \( r_2 \rightarrow u \)

C2: \( \) \( w'_1(v) \)

\( \rightarrow \) Msg. * is oblivious of \( r_2 \)

\( \rightarrow \) \( r'_2 \) must return \( v \)

**Faulty S:**

C1: \( w_1(u) \)

S: \( * \)

C2: \( r_2 \rightarrow u \)

C2: \( \) \( r'_2 \rightarrow v \)

\( \rightarrow \) S reorders \( w'_1 \) and \( r_2 \), as msg. * is oblivious of \( r_2 \)

\( \rightarrow \) If S forges same state at \( T \) as before, \( r'_2 \) returns \( v \)

\( \rightarrow \) History is not fork-linearizable