The Case for a Session State Storage Layer

Benjamin Ling and Armando Fox
Stanford University

Paper available in the back
Outline

- What is session state?
- Existing solutions, and why they are inadequate
- Proposed solution: Middle-tier storage layer
- Related and Future Work
What is Session State?

- Users interact with applications for a period of time, called a *session*.

- *Session state*—lifetime is the duration of a user session, relevant only to a single user.
  - User workflow in enterprise software
  - Shopping cart in eCommerce

- Many architectures produce/consume user session state (e.g., J2EE).

- *Session state is a large class of state*—we address a *subcategory*. 
An example of usage

Example of usage: Alice is using a web-based marketing application, specifying target customers and offers they should receive.
An example of usage

Example of usage: Alice is using a web-based marketing application, specifying target customers and offers they should receive.
An example of usage

Example of usage: Alice is using a web-based marketing application, specifying target customers and offers they should receive.
An example of usage

Example of usage: Alice is using a web-based marketing application, specifying target customers and offers they should receive.

Important:

- Session State must be present on each interaction
- Retrieval of session state is in critical path of app
Exploiting properties of session state

1. Not shared – user reads her own state
   - No concurrency control needed

2. Is semi-persistent
   - Temporal, lease-like guarantee is sufficient

3. Is keyed to a particular user – don’t need general query mech
   - Single-key lookup sufficient

4. Is updated on every interaction
   - Previous copies can be discarded

5. Does not need to be ACID
   - Only atomic update and consistency necessary
Existing solutions

- Database
- File System
- In memory, non-replicated
- In memory, replicated
Database or File System

“I already have a DB and FS, why don’t I just use one of them to store session state?”

Drawbacks:

- D1: Contention – Session state requests interfere with requests for persistent objects
- D2: Failure and recovery is expensive
  - Slow -> bad for end users
  - Fast -> usually very expensive
- D3: Session cleanup an afterthought
  - Someone has to scrub -> degrades performance
- D4: Performance hit – not just a round-trip, sometimes disk access
In-Memory: Replicated and non-replicated

- Try to avoid network roundtrip, usually faster than DB/FS
- Affinity: Require a user to “stick” to a particular server
- Middle-tier becomes stateful
- Affinity limits load balancing options
- Replicated -> pay RT cost, but usually not disk access
- Non-replicated -> lose data on crash
Replicated in-memory solutions: Drawbacks

- D5: Contention -> secondary App Servers now face contention from session state updates
- D6: Recovery more difficult -> special case code necessary -> system is harder to reason about
- D7: Poor failure/recovery performance
- D8: Lack of separation of concerns
  - App Server now does state storage and app processing
- D9: Performance coupling
Proposed solution: Middle-tier Storage

- Design principles
  - P1: Avoid special case recovery code
    - Reduces total cost of ownership
  - P2: Design for separation of concerns
  - P3: Session cleanup should be easy
  - P4: Graceful degradation upon node failures
    - No cache warming effects, uneven failure
  - P5: Avoid performance coupling
Assumptions

- Secure and well-administered cluster
- System Area Network (high throughput, low latency)
- No network partitions
- UPS reduces probability of system-wide simultaneous
- Fail-stop components
Middle-tier storage components

- Bricks: Stores objects via Hash table interface, periodic beacons
- Stubs: interface with bricks, keeps track of live bricks
Write Algorithm (Stub -> Brick)

- Call W the write set, WQ the write quota, R the read set.
- A stub:
  - Calculate checksum for object and expiration time.
  - Create a list of bricks L, initially the empty set.
  - Choose $W$ random bricks, and issue the write of \{object, checksum, and expiry\} to each brick.
  - Wait for $WQ$ of the bricks to return with success messages, or until $t$ elapsed. When each brick replies, add its identifier to the set $L$.
  - If $t$ has elapsed and the size of $L$ is less than $WQ$, repeat step 3. Otherwise, continue.
  - Create a cookie consisting of $H$, the identifiers of the $WQ$ bricks that acknowledged the write, and the expiry, and calculate a checksum for the cookie.
  - Return the cookie to the caller.
Write example

Try to write to W bricks, \( W = 4 \)
Must wait for WQ bricks to reply, \( WQ = 2 \)
Write example

Try to write to $W$ bricks, $W = 4$
Must wait for $WQ$ bricks to reply, $WQ = 2$
Write example

Try to write to $W$ bricks, $W = 4$
Must wait for $WQ$ bricks to reply, $WQ = 2$
Write example

Try to write to $W$ bricks, $W = 4$
Must wait for $W_Q$ bricks to reply, $W_Q = 2$
Read Algorithm (Stub -> Brick)

- Verify the checksum on the cookie
- Issue the read to $R$ random bricks chosen from the list of $WQ$ bricks contained in the cookie.
- Wait for 1 of the bricks to return, or until $t$ elapses.
- If the timeout has elapsed and no response has been returned, repeat step 2. Otherwise, continue.
- Verify checksum and expiration. If checksum is invalid, repeat step 2. Otherwise continue.
- Return the object to the caller.
Read example

Ask R bricks for the read, wait for fastest 1 to reply. \( R = 2 \)
Read example

Ask R bricks for the read, wait for fastest 1 to reply. R = 2
Ask R bricks for the read, wait for fastest 1 to reply. R = 2
Read example

Ask R bricks for the read, wait for fastest 1 to reply. R = 2
What happens on failure?

- All components stateless/soft-state
  - Restart!

- App Server Crash
  - Restart and reconstruct list of live bricks

- Brick Crash
  - All state on brick is lost, but...
  - Copies are in \( W \) other bricks, so state is not lost
  - Rejuvenation of state

- Easily add new nodes to a running system
Interesting properties

- Negative feedback loop example
  - Let write group for a given write be \( A, B, C \). \( B \) is slow.
  - \( WQ = 2 \)
  - Since \( B \) is slow, will not reply to write of a key \( X \)
  - \( B \) won’t be involved in read of \( X \)
  - May help ease load on overloaded nodes

- Bricks can say “no”
  - Since more writes are issued than necessary, when overloaded, a brick can drop writes
Related Work

- Quorums – don’t need to read so many copies, since we know where up-to-date copies live
- DDS – performance coupling, persistent, negative cache-warming effects
- “Directory oriented available copies” – uses a directory to find available copies
- Berkeley DB – emphasized fast restart and failure as a common case
- DeStor – focuses on persistent storage
Conclusion

- Session State Server that:
  - Performs well, without coupling
  - Fault-tolerant
  - Recovers instantly
  - Scalable
  - Lowers total cost of ownership
Questions?

bling@cs.stanford.edu

http://www.stanford.edu/~bling/SessionStore.ps