History of a cache

An experience report

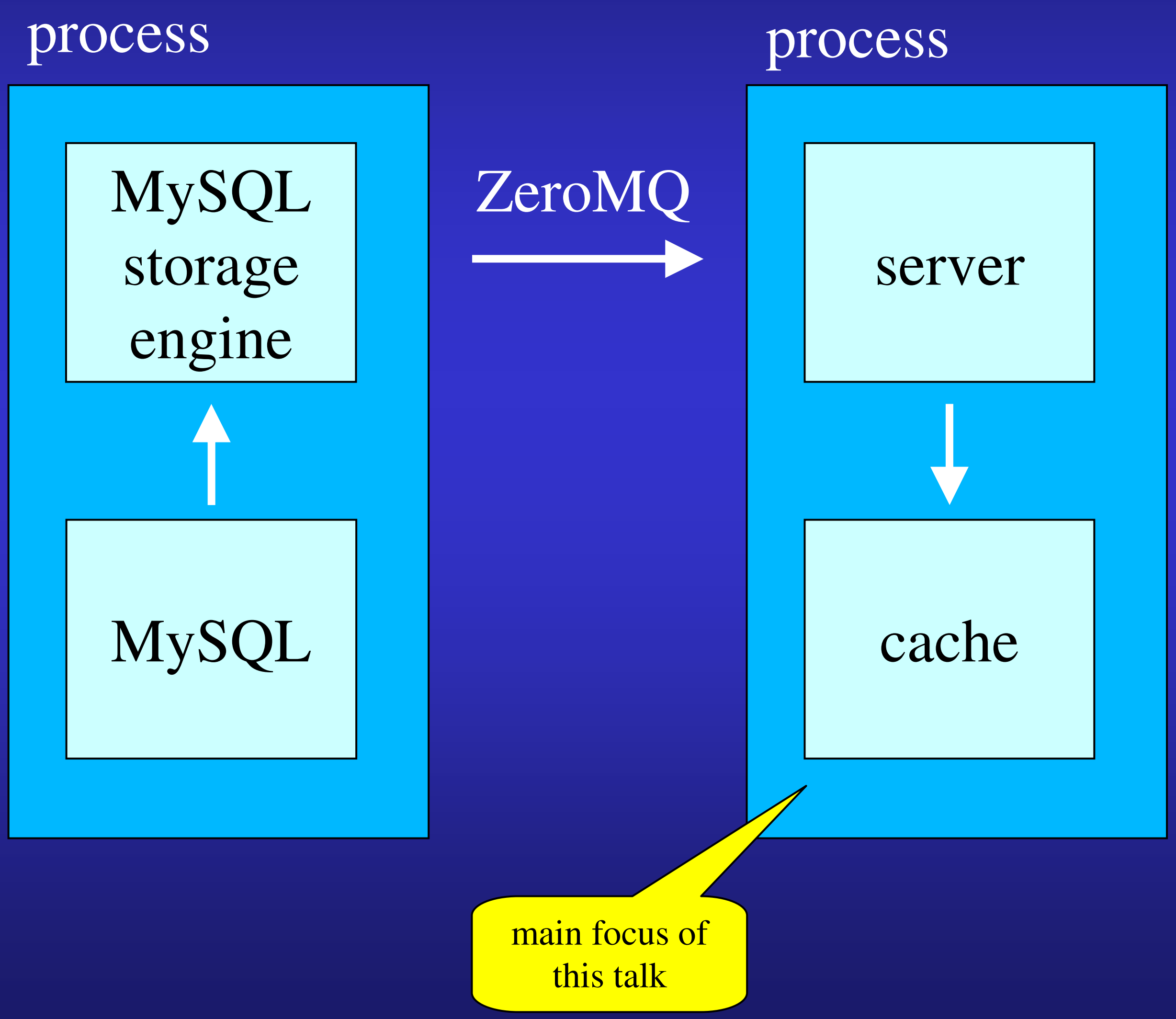
Hubert Matthews
ACCU 2013
hubert@oxyware.com

April 2013

Overview

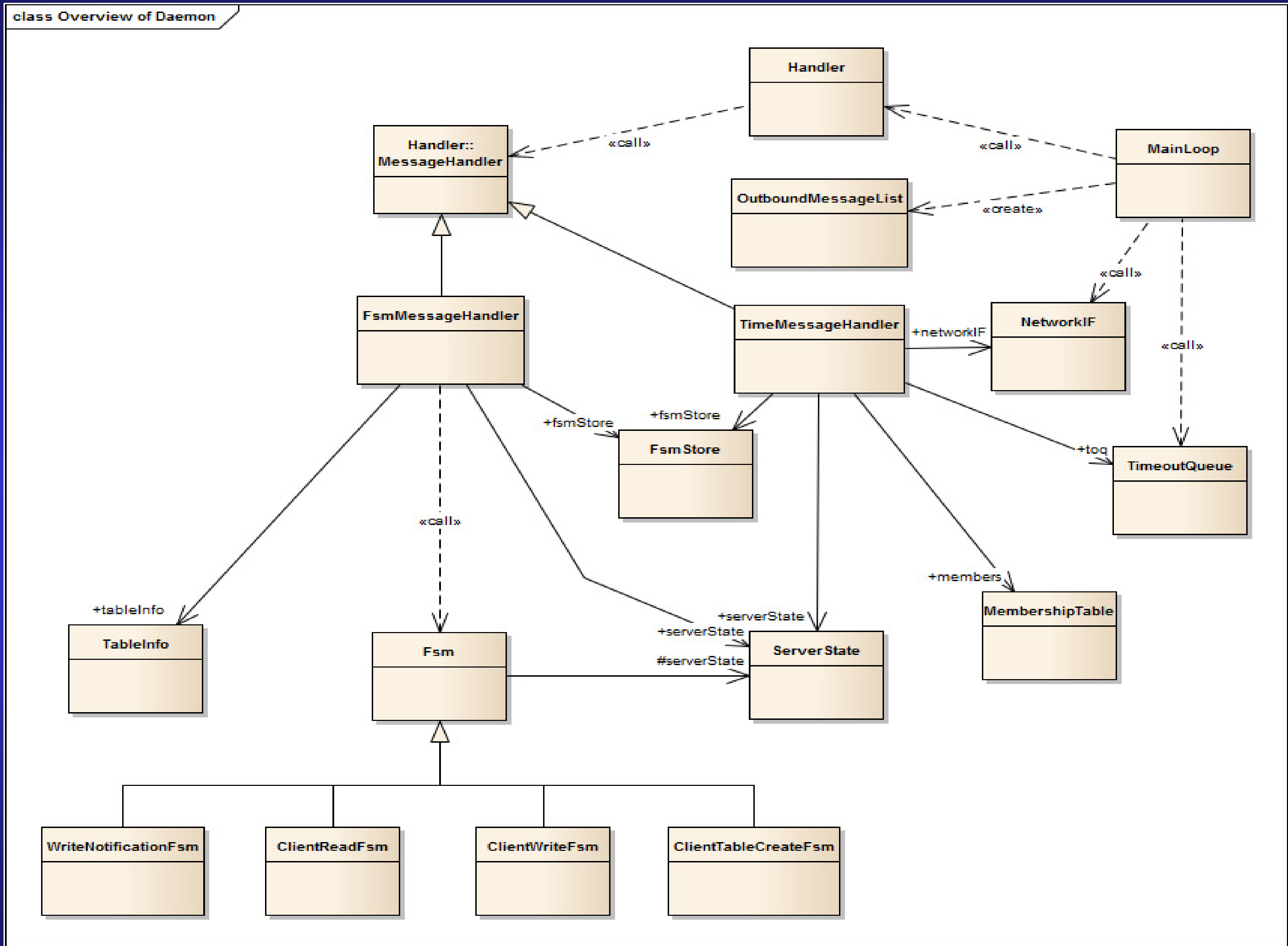
- A mixture of issues from a real project
- The trials and tribulations of TDD
  - when it's good
  - what to do when it's not
- Dealing with significant requirements change
- Interface design and concurrency issues
- Incremental change
- The endless need for speed

Original architecture



Original server design – the good bits

- Built in modern C++ with TDD from the start
- Only one class that talks to the environment
  - sending and receiving network messages
  - in the main polling loop and calls all the other classes
- Everything else in-memory and single threaded
- Dependency injection – c/trs wire up objects only
- 94% line coverage using GoogleTest and gcov/lcov
- 450 unit tests that run in < 1 sec (on file save)
- State machines per message type + FSM base class
- Cache based on std::map and iterators
- 5K LOC server + 6K LOC unit tests


class Overview of Daemon
Handler
Handler::
MessageHandler
«call»
«call»
MainLoop
OutboundMessageList
«create»
«call»
FsmMessageHandler
TimeMessageHandler
+networkIF
NetworkIF
«call»
+fsmStore
+fsmStore
FsmStore
+toq
TimeoutQueue
«call»
+members
MembershipTable
+tableInfo
TableInfo
Fsm
+serverState
+serverState
#serverState
ServerState
WriteNotificationFsm
ClientReadFsm
ClientWriteFsm
ClientTableCreateFsm

State machine base class

```
class Fsm {
    struct StateFuncProxy;
    typedef StateFuncProxy (Fsm::*FuncPtr)(const Message & msg);

    struct StateFuncProxy {
        StateFuncProxy(FuncPtr pp) : p(pp) {}
        operator FuncPtr() const { return p; }
        FuncPtr p;
    };

    virtual StateFuncProxy initState(const Message & msg);
    virtual StateFuncProxy finalState(const Message & msg);

public:
    StateFuncProxy currentState;        // what state we are in currently
    int count;                          // internal state of FSM

    Fsm() : currentState(initState), count(0) {}

    void handleMessage(const Message & msg) {
        currentState = (this->*currentState)(msg);
    }

    bool isInFinalState() const {       // can we delete this FSM?
        return currentState == &Fsm::finalState;
    }
};
```

Original architecture – the good bits

- Keep legacy MySQL code separate from server
  - No RTTI or exceptions
- Message-driven approach allows for easier debugging, tracing, system testing, support
- All business logic in server so unit testable
- Asynchronous and event-driven for performance
- Performance was expected to be disk limited because of the use of caching
- Original version worked well and good development progress

…then things changed…

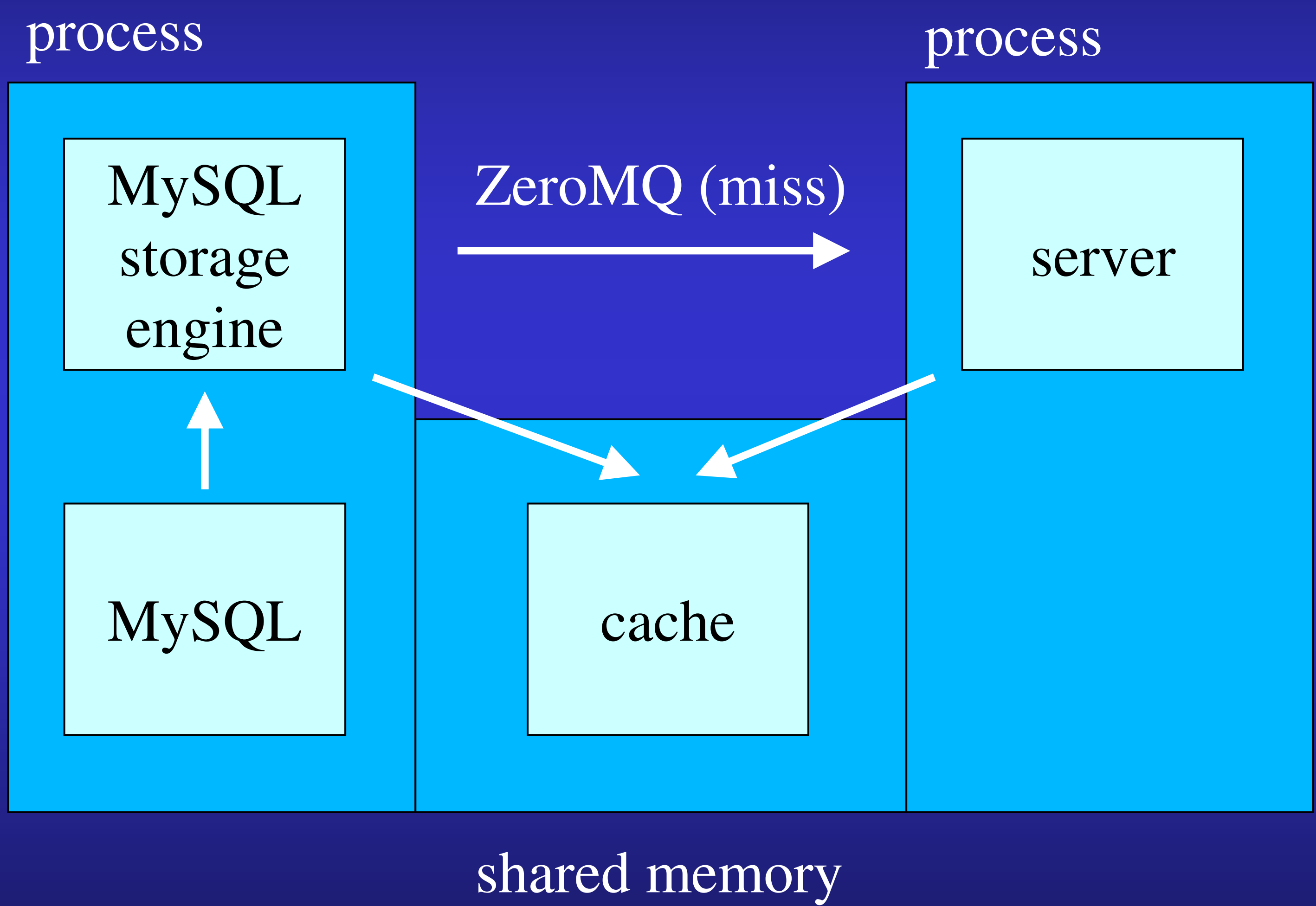
The problem and options

- Speed now became a major requirement
- Benchmark results
  - 10-20x too slow on simple single-threaded queries
  - even slower than that on complex range queries
- Options
  - start again with a different architecture?
  - migrate to a new architecture?
  - tune our way out of the problem?
  - go multi-threaded?
- Time for some measurements….

Thoughts and experiments

- Is it messaging overhead?
  - tests with ZeroMQ between threads (12K/sec), shared memory queue (57K/sec), Boost message queue (53K/sec) and spin loops on an atomic (2,000K/sec)
- For all but the spin loop there were 4 context switches per message (synchronous protocol)
  - the spin loop burns CPU but has great throughput
- Therefore, avoid context switches for speed
  - original design used asynchronous messaging protocol
- This implies we need to use MySQL's threads directly to query the cache

New architecture



Implications of new architecture

- Cache now needs to be in shared memory
  - but std::map won't work directly in shared memory
  - use a custom allocator for std::map?
  - build a std::map equivalent?
  - lifetime management of cache entries?
- Cache now needs to be thread safe
  - will adding locks be sufficient?
  - how to handle concurrent modifications?
- How to unit test concurrent code?
  - TDD and concurrency?

New architecture – first steps

- Use boost::interprocess::map, string and friends
- Provides a cross-platform std::map equivalent mapped into shared memory
- This solves some of the shared memory issues but not the concurrency ones
- Just adding internal locking to all the cache's methods is not sufficient
- std::map has the wrong interface for concurrency!

Concurrency and std::map interface

```
// standard std::map usage pattern

MapIterator i = cache.find(key);

if (i != cache.end()) {

    doSomethingWith(i->first, i->second);
}
```

what if someone changes the cache here and cache.end() changes?

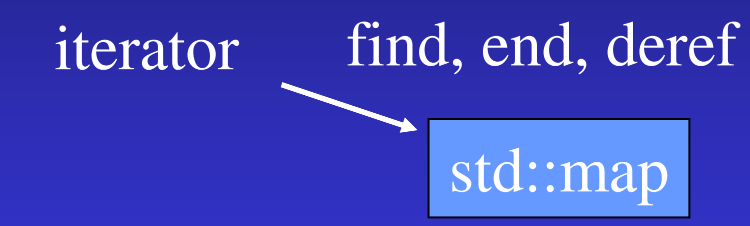
what if someone deletes or changes the item you just retrieved?

- Standard usage for std::map has two race windows
- Locking each operation individually doesn't help
- What are the options?

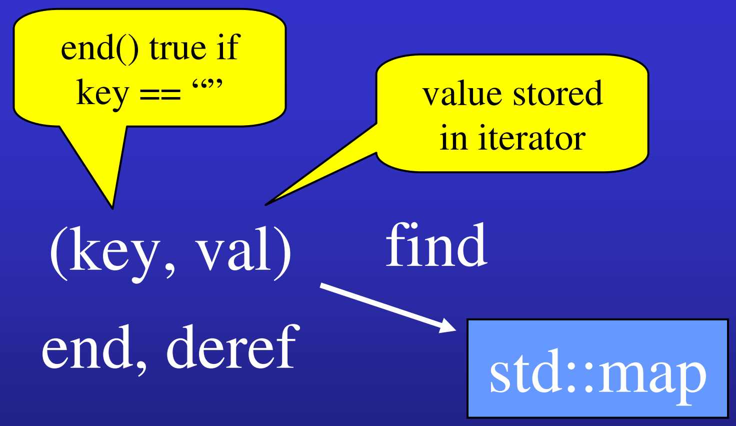
Cache interface options

- Lock the entire cache over all three operations
  - kills performance by serialising access to the cache
  - relies on all callers doing so explicitly (one offender is sufficient to cause nasty intermittent bugs)
  - requires changing all of the client code
- Change the cache interface to a race-free interface
  - requires changing all of the client code and unit tests
- Changing the cache interface breaks all the unit tests
  - refactoring in the dark with no safety net
- Is it possible to support this interface in a parallel-friendly fashion? Yes!

Race-free std::map compatible interface

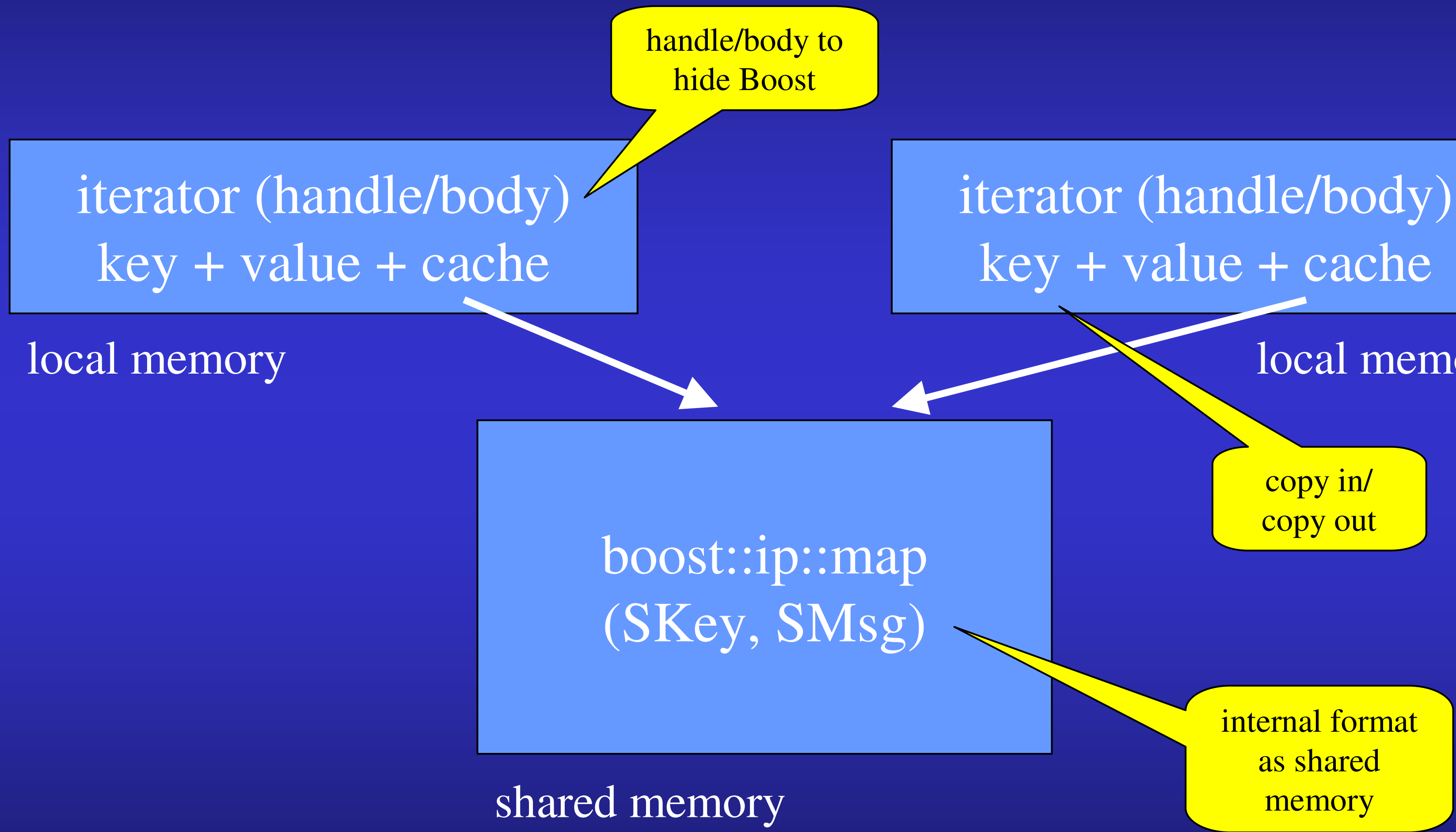


Original std::map i/f refers to the cache for all three operations



New “thick” iterator with key+value is returned by (locked) find; end and deref do not refer to the cache at all so no race windows

Race-free std::map details



Shared memory aspects

- Needed an anti-corruption wrapper to hide the shared memory implementation
  - MySQL can't compile Boost so had to wrap boost::interprocess::*
  - shadow classes: SharedMsg wrapped Message
- This implied a copy-in/copy-out approach for all cache accesses as we couldn't refer directly to the values in the cache
  - we gained a factor of 10% in speed, only 20x to go…
  - essentially replaced a messaging layer with copying data
- All of the unit tests passed without change
- No change to client code either
  - typedef for cache allowed fast switching of implementations

Testing concurrent code

- How do you know it's correct?
  - just by chance because you didn't find the race yet?
  - run unit tests in parallel
- Use GoogleTest repeat and shuffle functions to run unit tests 100 times in random order
- Use linux xargs to run processes in parallel to check exclusion works
- `echo $(seq 1 10) | xargs -P 0 -n 1 ./test -gtest_repeat=100 -gtest_shuffle`
- Shared memory causes startup issues when running parallel
  - can't start from zero state – have to use shared state
- And still there was a concurrency bug….(more on that later)

Performance counters

- Added performance counters
  - stats, visibility, debugging
  - very useful for checking cache hits and misses
- Cache aligned padded struct – name + atomic 64-bit counter
- Static reference to each counter in cache operations
  - `static Utils::Stats::Counter & statsInsert = Utils::GetStats().findCounter("cache.insert");`
- Memory-mapped file plus dump program
- Allows real-time monitoring of updates
- `$ watch -n 1 -d ./cache-dump stats-file`
  - highlights changing counters on a per second basis
- Using performance counters highly recommended

Counter oddities

- Counter for cache memory use was massively large at times
- Memory overwrite? Bug? Counter issue?
  - But we're using TDD it can't be a bug, or can it?
- Added invariant assertion to cache update methods
  - assert(invariant()); - class invariants + data sanity check
- Found a bug!
  - `cacheSize += size(newElem) - size(oldElem);`
  - cacheSize is unsigned and size calculation is approximate so can go negative – oops!
  - added sanity check on cache size update
- Assertions to check invariants highly recommended

Key comparison speed

- The key for the cache was a three-field composite key: (database, table, key)
- Benchmarks showed that op< for this key used a significant portion of the time

```
bool operator<(const SharedGKey & gkey1,
               const SharedGKey & gkey2)
{
  return gkey1.database.compare(gkey2.database) < 0
      || (gkey1.database.compare(gkey2.database) == 0
          && gkey1.table.compare(gkey2.table) < 0)
      || (gkey1.database.compare(gkey2.database) == 0
          && gkey1.table.compare(gkey2.table) == 0
          && gkey1.key.compare(gkey2.key) < 0);
}
```

More speed, please…

- Refactoring the key to be a single std::string with the three fields separated by NUL characters allowed for simpler and faster key comparison
- A factor of 3x faster lookup

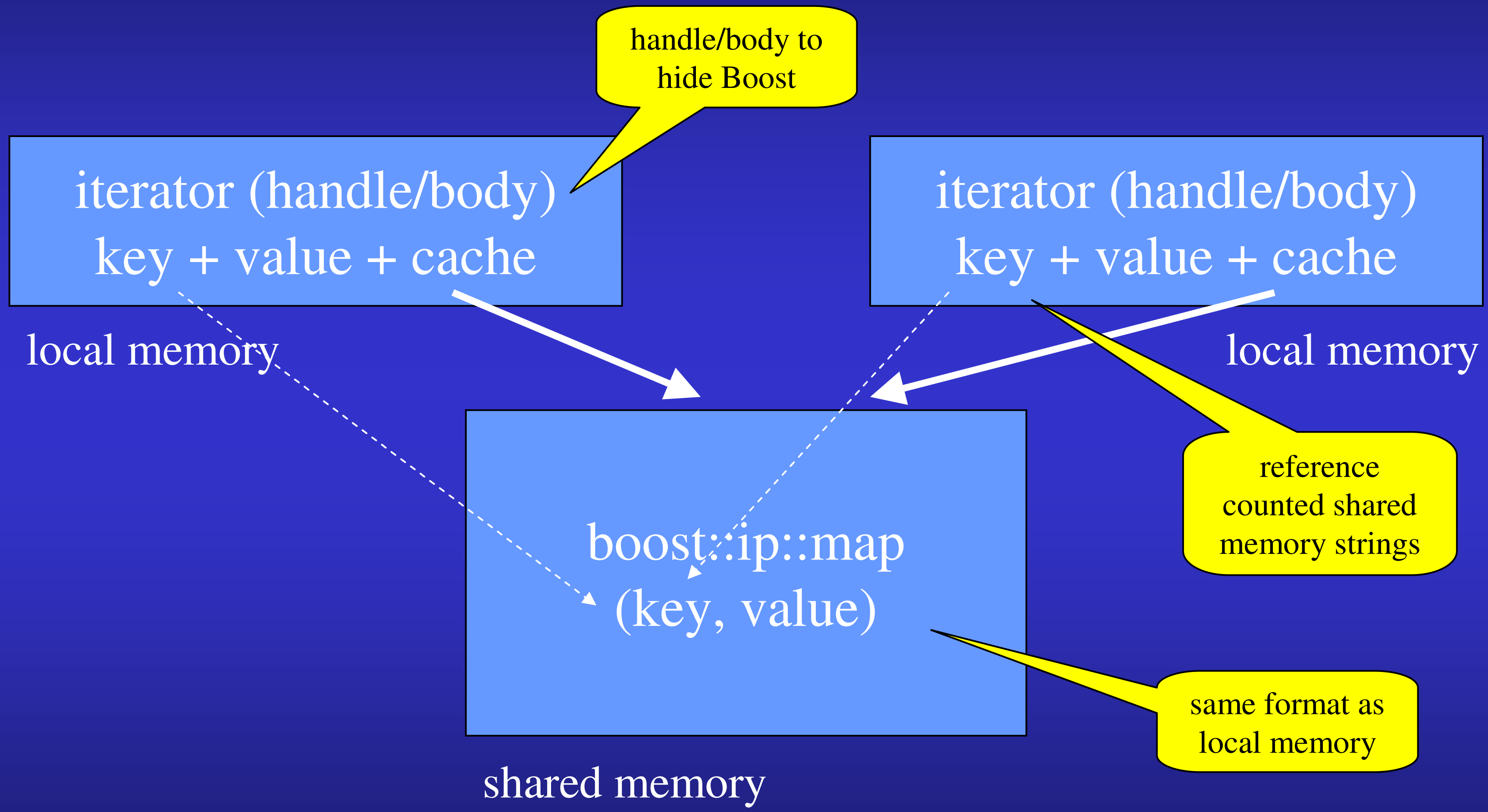
```
GlobalKey(const Database & database,
          const Table & table, const Key & key)
{
    composite  = database;  composite += '\0';
    composite += table;     composite += '\0';
    composite += key;
}

bool operator<(const GlobalKey & other) const {
    return composite < other.composite;
}
```

Shared memory and fixed addresses

- Shared memory mapping into different processes means that internal native pointers don't work
  - mapped to different virtual addresses in each process
  - need to use offset (relative) pointers instead
- Slower dereferencing as can't use native pointers
- Use fixed-position shared memory segment in Boost
  - gave a factor of 2x in speed as it can use native pointers instead of relative pointers/offsets
- Still copying data in and out (potentially large)
- Using native pointers internally means we can use shared addresses to avoid copies

Shared data


handle/body to hide Boost
iterator (handle/body)
key + value + cache
local memory
iterator (handle/body)
key + value + cache
local memory
reference counted shared memory strings
boost::ip::map
(key, value)
same format as local memory
shared memory

Shared immutable cache items

- Immutable cache items means that they can be shared between threads with no copy-on-write or other fancy stuff that would require locking
  - reference counting means that even if an item is deleted from or overwritten in the cache it is still accessible
- Only locking is on atomic incr/decr of reference count
  - use gcc intrinsics for atomic operations
- Used Boost shared memory allocator plus own simple ShmString that allocates all memory for string plus length and ref count in one allocation
- A factor of 3x in speed – we're looking better!

Shared immutable cache items - detail

```
class ShmString {
public:
  ShmString() { rep = 0; }
  ShmString(const char * p, uint32_t len) {
    void * where = SharedMemory::allocate(sizeof(Rep) + len);
    rep = new (where) Rep(p, len);
  }
  ShmString(const ShmString & other) : rep(other.rep) {
    if (rep) __sync_fetch_and_add(&rep->refCount, 1);
  }
private:
  struct Rep {
    Rep() : refCount(1), size(0) { body[0] = '\0'; }
    Rep(const char * p, uint32_t len) : refCount(1), size(len) {
      memcpy(body, p, len);
    }
    uint32_t refCount, size;
    char body[1];             // will be longer than this....
  };
  Rep * rep;
};
```

Mixing native iterators and find

- Range queries require iterator++ to work
- op++ is a simple and fast operation on a single-threaded cache such as std::map (internal pointer)
- How to support op++ on a concurrent cache? races!
- Using upper_bound to find next key in cache is slow (cache iterator uses key by value, not iterator)
- Solution: use a composite iterator with (key, value) pair plus a native iterator and a cache sequence number
- op++ performed by the cache when locked and uses native iterator if seq num not changed or upper_bound if it has

Mixing native iterators – details

- Cache iterator = key + value + iterator + seq num

```
void Cache::insert(Key key, Value value) { /*…*/ seq_num++; }

iterator Cache::findNext(iterator i) {
  if (i.seq_num == seq_num)
    return ++(i.native);
  else {
    return findNextBasedOnKey(i.key);
  }
}
```

- Factor of 4x-5x in speed for range queries with no concurrent updates

Cache eviction policy

- The cache uses a least-recently used eviction policy
- This is implemented using a std::list and when a node is looked up through the std::map interface the node is moved to the head of the list
- When a node needs to be evicted then the last node on the std::list is erased and its key removed from the lookup map
- This policy caused too many new/delete pairs so LRU eviction was turned off if the cache size was less than the max cache size
- Performance gain estimated at 20%

```
/usr/lib/libcloudfabric.so.0(Daemon::SharedCache<Messaging::GlobalKey,
Messaging::MsgHandle, std::less<Messaging::GlobalKey>
>::evictOldestElement()+0x2a4) [0x7f50e6d0ea14]
 /usr/lib/libcloudfabric.so.0(Daemon::SharedCache<Messaging::GlobalKey,
Messaging::MsgHandle, std::less<Messaging::GlobalKey>
>::SharedCache(int)+0x2ef) [0x7f50e6d11a3f]
 /usr/lib/libcloudfabric.so.0(Daemon::RowCache::RowCache(unsigned
long)+0x31) [0x7f50e6d0b931]
/usr/lib/libcloudfabric.so.0(cf::CloudFabric::Impl::Impl(zmq::context_t&
, std::string const&)+0x1a1) [0x7f50e6cff3e1]
 /usr/lib/libcloudfabric.so.0(cf::CloudFabric::CloudFabric(cf::Context&,
std::string const&)+0x3c) [0x7f50e6cf4fcc]
 /usr/lib/libcloudfabric.so.0(cf_connect+0x68) [0x7f50e6cf5068]
/usr/lib/mysql/plugin/ha_geniedb.so(geniedb::genieHandler::getConnection
()+0x29) [0x7f50e6f49fd9]
/usr/lib/mysql/plugin/ha_geniedb.so(geniedb::genieHandler::connection()+
0x59) [0x7f50e6f4a1a9]
/usr/lib/mysql/plugin/ha_geniedb.so(geniedb::genieHandler::getRecordCoun
t()+0x30) [0x7f50e6f4a200]
/usr/lib/mysql/plugin/ha_geniedb.so(geniedb::genieHandler::info(unsigned
int)+0xb8) [0x7f50e6f4c578]
 /usr/sbin/mysqld(+0x3eb010) [0x7f50ebd30010]
 /usr/sbin/mysqld(JOIN::optimize()+0x50d) [0x7f50ebd324ed]
 /usr/sbin/mysqld(mysql_select(THD*, Item***, TABLE_LIST*, unsigned int,
List<Item>&, Item*, unsigned int, st_order*, st_order*, Item*,
st_order*, unsigned long long, select_result*, st_select_lex_unit*,
st_select_lex*)+0xd7) [0x7f50ebd35b57]
 /usr/sbin/mysqld(handle_select(THD*, st_lex*, select_result*, unsigned
long)+0x174) [0x7f50ebd3b524]
```

Concurrency bug

- After ½ hour loading in 2GB of data
  - `/usr/lib/libcloudfabric.so.0(Daemon::RowCache::RowCache(unsigned long)+0x31) [0x7f50e6d0b931]`
- Why is the code still in the c/tr after ½ hour?!
  - Clients create and delete cache objects per connection
- The c/tr wasn't locked
  - normally c/tr isn't locked as you can't share an object until it's been created
  - not true for stateful objects with shared memory!
- 10 mins to fix the problem, 4 hours to reproduce it

Version control branching

- All these cache modifications were made on trunk
- Allows for early feedback on performance and integration issues
- Branching in VC is a design smell
  - caused by semantic differences
- VCs merge syntactically and not semantically
- Don’t branch unless you have too
  - simple incremental changes or refactor until you can
- Don’t take off unless you know where you’re going to land!

Lessons learnt

- TDD works best when you design for testability
  - critically dependent on the quality of tests so get good at test thinking
- Assumptions about requirements need to be validated – caveat architect!
- Incremental changes are preferable to major change
  - relative progress may seem slower at the time but the overall arrival time is shorter (and less stressful)
  - branch as a last resort and plan your return trip carefully
- Concurrency correctness is tricky to unit test
- Immutable data makes sharing easier

Lessons learnt (cont'd)

- Assertions that check class invariants and data items are well worthwhile
  - make it triggerable at runtime
- Performance counters really help to understand a system's behaviour
  - help developers as well as testers and operators