Database Operations Using Asynchronous Remote Method Invocation

James Sproch

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-263
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-263.html

December 14, 2012

Database Operations Using Asynchronous Remote Method Invocation

Jim Sproch
jsproch@{stanford, berkeley}.edu
Stanford University & UC Berkeley

Abstract

Today's highly scalable Internet applications are largely bound by speed-of-light communication delays between geographically disparate data centers. This latency is particularly problematic when writing to distributed database systems because waiting for messages to travel halfway around the world is often unacceptable, and maintaining any notion of consistency is exceedingly difficult when using modern asynchronous architectures. We present a novel architecture, based on asynchronous remote method invocation, that achieves interesting consistency guarantees, even when performing write operations asynchronously. An analysis of our implementation demonstrates that this solution can provide an intuitive interface to developers, and doesn't require any specialized knowledge of databases, SQL (Standard Query Language), or transactions.

Introduction

The computational resources required for a single application to service millions of Internet users can easily exceed the capacity of a single data center. This means applications must scale horizontally across multiple data centers. Service providers have found it useful to keep these data centers geographically separated to minimize the distance between a user and the nearest data center, as well as to maximize the reliability and availability of the service by keeping failures independent. This separation, however, makes communication between data centers very expensive. A single round-trip message between two data centers can be 100 milliseconds or more, and is lower bounded by the speed of light.

Avoiding these expensive round-trip calls, therefore, has been a top priority for service providers. The vast majority of such calls are read/write operations on data. To minimize expensive reads, data can be cached and replicated across clusters. Minimizing expensive writes is more difficult because there can be a variety of race conditions when two writers act simultaneously.

Formally, a 'write conflict' is a race condition that occurs when a client attempts to commit changes to a record when the changes are based on an out-of-date read. This may occur when two (or more) clients are attempting to perform an operation at the same time. They both read, they both perform a computation, and both try to write their results. Since the second writer calculated his result before the first writer's update was available, the second writer's result is now potentially invalid or incorrect.

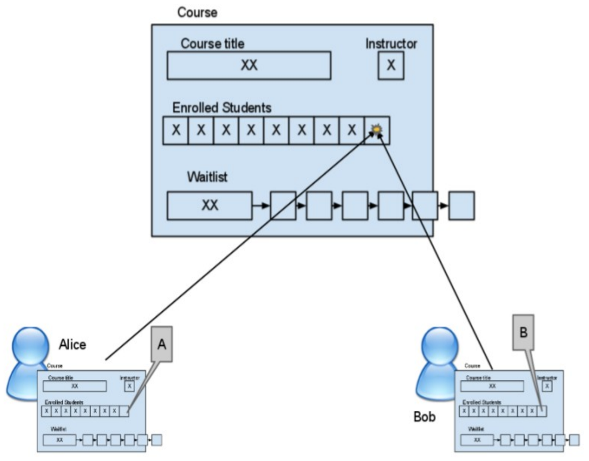


Figure 1: Alice and Bob both to enroll(asynchronously) In a course, but the two writes conflict.

If we assume the round trip latency (~100ms) of a synchronous write (waiting for the database server to respond with ‘ok, commit successful’ before proceeding) is unacceptable, we are left with four known options for resolving write conflicts:

1. Raise a callback when a write fails. The database server notifies the client that the operation failed; then, the client is expected to execute a user-defined callback handler, which is responsible for undoing any work that depended on the write and/or retrying the operation. In practice, the retry logic to handle such callbacks may be fairly complicated and is rarely implemented correctly by application developers.
2. Use a default conflict resolution policy. This generally is “last-write-wins”, and may result in some updates being lost due to race conditions. If the writes only partially overlap, this policy could also lead to an inconsistent state.
3. Write now, consolidate later. This solution saves both versions, and later applies a user-provided consolidation function that attempts to merge the conflicting versions. This requires that, given two conflicting copies of the data, there are easy rules to reconcile the differences. The creation of generalized reconciliation rules, however, is currently an unsolved problem.
4. Push application logic into the transaction. Simple modifications can oftentimes be encoded into a SQL update operation, which performs the read-update-write logic in a single operation. Other more advanced/complicated domain specific languages (DSLs) can allow for more sophisticated application logic to be embedded into the transaction, but often creates maintenance and portability concerns. By delaying the decision of how to manipulate the data, transactions can be applied in a serializable order, and a reordering of these transactions will merely change the behavior of the later transactions.

We present an application of the fourth technique -- improving on existing approaches by utilizing asynchronous remote method invocation. Prior to this work, pushing application logic into a transaction required special-purpose transaction description languages. We show that the same benefits can be achieved without requiring the user to write any additional code. Furthermore, our technique makes it easy for developers to get a reasonable prediction about the expected result, and use that prediction until a committed result is available from the database server. We find that these predictions are sufficient approximations of reality to allow most computations to continue without waiting for a response from the database server.

We will start by examining the current techniques used in modern systems to avoid round-trip latency, followed by a presentation of our own technique (asynchronous remote method invocation with local execution). We then detail our prototype implementation of the technique, and describe some of our design decisions. Lastly, we present a few performance benchmarks that demonstrate some of the advantages of using asynchronous remote method invocation.

State of the Art

Dealing with concurrency control in distributed database systems is not a new problem; it has been a problem since at least the 70’s [1]. More recently, database researchers have started building on the premise that it may be beneficial to exchange strict transactional consistency for performance, availability, or predictability [2], and generally refer to the eventual propagation of data to weakly consistent replicas [5]. Most of this research assumes a last-write-wins policy as implemented in SCADS [3], but the research community is still considering other alternatives including the accumulation of messages as demonstrated in the disorderly shopping cart discussion [4]. Systems like Dynamo allow users to choose a level of consistency versus performance, enabling optimization based on the factors deemed most important [6].

Regardless of the consistency policy, the shift to world-wide distributed networks has prompted substantial interest in reducing the latency of round-trip messages to remote servers. For instance, protocols like MDCC attempt to reduce the number of round-trip messages exchanged between data centers by building on optimizations of the Paxos consensus algorithm including Fast Paxos and Generalized Paxos [7]. Many of these systems, including the MDCC programming model provide callback handlers that inform the application of a transaction's end state (success/failure), allowing the application to react accordingly. The key downside to these systems, however, is that it's generally difficult for an application to make forward progress until receiving the responses (thus necessitating at least one round trip).

Asynchronous Remote Method Invocation (ARMI)

Our technique (ARMI with local execution) allows an entity to be modified through a series of method invocations, which are executed locally on a cached copy and asynchronously on the primary copy. These methods will typically be setter and getter methods, responsible for providing access to the data structure, and maintaining internal integrity by performing the appropriate checks and input validations. This encapsulation is standard practice when writing Java programs, and we leverage the practice to provide natural primitives for manipulating data within an entity graph

When a method is invoked, the method actually runs twice. First, the method runs locally on the client machine, producing a result based on the entities in the client-side cache. The local execution provides an approximate prediction of the 'true' result. Assuming the method is deterministic, and the cache is up-to-date, the prediction will be exactly correct. Even if the cache is out of date, the result of a local execution is often 'good enough' for the application to continue; a claim we demonstrate in the examples below. In the second step, an asynchronous message is sent to the database server, informing the server of the method invoked and the arguments used. The server will proceed to perform the same operation on the primary copy, and will send the updated version back to the client, who can then update the client-side caches.

To better understand this solution, we consider a couple of examples.

Incrementing Request Count

One of the simplest entities is a counter responsible for keeping track of how many times a particular event occurred. Suppose, for instance, we want to put one of those view-count tickers at the bottom of the page, tracking the number of times a given web page was requested. We have a cluster of these machines, all serving the page, so we may have many concurrent writers trying to increment the count.

If each client did a read, an increment, and a write, we might lose some of the counts. Two clients would do a read, after which, they would both do an increment, followed by a write. Both clients would be writing the same value (n+1), even though two web requests had come in (one from each server).

This race condition is eliminated when we use remote method invocation to invoke a counter.increment() method. For the purposes of this example, let us suppose counter.increment() returns the current count. Two servers invoke this method (asynchronously) using remote method invocation. Each client runs the method locally, and the method returns a prediction of (n+1) for each of the clients. This is a reasonable guess, and for the purposes of continuing the computation, it's probably good enough. Now, both messages arrive at the database, and the database executes both of them. The first one will return (n+1) and the second one will get a value of (n+2). The new final value (n+2) will be propagated back to the servers, which will continue incrementing from this new value.

Eventually, everyone gets the updates from the database, and everyone converges to the correct value. None of the counts are ever lost, despite the race condition. Some of the web servers may get a slightly old/inaccurate view of the count due to caching, but the guess is good enough to continue rendering the page. It's certainly better than waiting an additional 100ms for a round trip call to the centralized database.

The naive implementation of this counter is shown in Appendix A1. There are much better ways to implement such a page counter, and there are better ways to do this, but it makes for a simple and understandable demonstration.

Linked List
A more sophisticated example is a custom doubly-linked-list data structure. Each node in the list has a next node and a previous node. The head node has a reference to the tail node. Possible steps to append to the list are:

1. Create a new tail node (newTail)
2. set newTail->previous = head->tail
3. set newTail->next = null
4. set head->tail->next = newTail
5. set head->tail = newTail

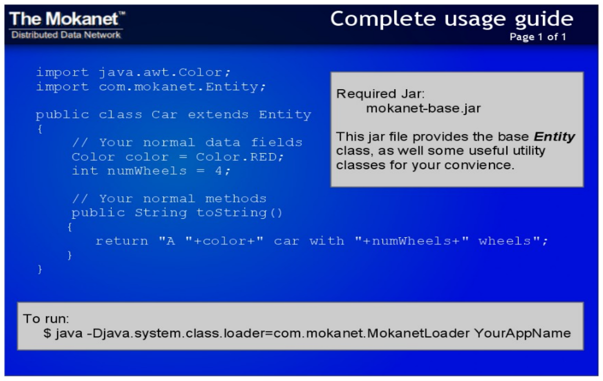
A race condition in this sequence would be disastrous. Fortunately, with remote method invocation, list.append() will occur atomically. At the end of a race between two writers, we will always have correctly appended two additional nodes.

The predictions, however, are reasonable. Each writer sees a new list with the respective new datum appended to the end. Modulo a minor reordering of entries, the predictions are correct. The database server response will update the list in the second client's cache to reflect the fact that the first writer's insert actually preceded the second writer's insert.

When predictions aren't good enough
One can imagine circumstances, including most financial transactions, where an educated guess is not sufficient, and the application may need confirmation before making forward progress. In such cases, no asynchronous technique is going to sufficiently meet the design requirements, and the round trip database latency is unavoidable. Our implementation does support this mode of operation with an optional annotation (@Remote), but we argue that this annotation should be applied sparingly. In the vast majority of applications, such strong consistency isn't necessary or practical in large, scalable services.

Assumptions & Justification
Most Internet workloads are read-heavy, so despite the potential for conflicts, they're actually somewhat rare. In the common case (no conflict), the prediction will be correct, and orders of magnitude faster - as shown in the 'performance metrics' section. Even when the prediction is incorrect, it should be reasonably close to the truth, and will converge to to a consistent solution. Since many applications are shifting to eventual consistency databases, and our solution is strictly superior to the majority of techniques used by other eventual consistency databases, we believe this approach warrants further research.

The Mokanet Datastore



The Mokanet™
Distributed Data Network

Complete usage guide
Page 1 of 1

```
import java.awt.Color;
import com.mokanet.Entity;

public class Car extends Entity
{
     // Your normal data fields
     Color color = Color.RED;
     int numWheels = 4;

     // Your normal methods
     public String toString()
     {
        return "A "+color+" car with "+numWheels+" wheels";
     }
}
```

Required Jar:
mokanet-base.jar

This jar file provides the base ***Entity*** class, as well some useful utility classes for your convience.

To run:
```
$ java -Djava.system.class.loader=com.mokanet.MokanetLoader YourAppName
```

The simple object-oriented interface means a user doesn't need any prior database knowledge. The entire usage guide fits on a single GooglePresentation slide.

The Mokanet is a distributed datastore for Java, that takes the place of more traditional database solutions like MySQL. Unlike MySQL, the Mokanet is a NoSQL datastore built on top of a shared-nothing key-value architecture, and uses asynchronous remote method invocation as described above. The persistence API (application programming interface) is completely transparent to the application developer, allowing programmers to start using the system without any training.

Object-Oriented Programming

Developers work in the completely object-oriented world provided by the underlying language. Instantiating new objects will result in new records being created in the database. Modifying these objects will cause equivalent changes within the database. When all references to an object have been eliminated, the underlying record will be garbage collected, just like standard objects within the Java Virtual Machine (JVM). Users are provided with a variety of standard data structures: lists, sets, hash maps, etc. Users can also extend these data types, or define custom data structures as usual.

Entities

Entities are the primitive data types of the system, and should be plain old Java objects (POJOs). They should consist of data fields and simple setters & getters; they should not be performing complex computations or performing privileged operations. To indicate to the system that a particular object should be persisted, we require that the data class extend the type `com.mokanet.Entity`. All data entities should extend this base class, or have an ancestor that extends the `Entity` class. Entities have some additional restrictions, such as only being permitted to contain references to other entities, but users can learn these rules quickly without prior knowledge. These additional restrictions will be described in detail later.

Object Graph

Just as in Java, programmers view the world as a large interconnected graph of objects (entities). Each object may have references to other objects, and the graph can be traversed programmatically. The size of the graph may far exceed the system's physical memory, so nodes are lazily swapped in and out of memory as needed, giving the illusion that the entire graph is always in memory. Furthermore, the entire graph is automatically persisted in the datastore. Users can get a static

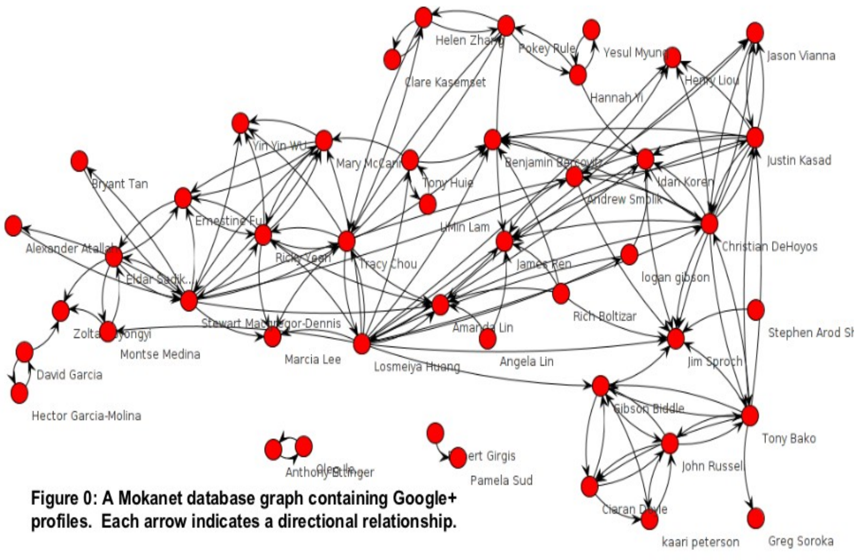


Figure 0: A Mokanet database graph containing Google+ profiles. Each arrow indicates a directional relationship.

reference to this object graph by using a static variable within the class definition of one of their entities. Because all the fields within an entity are persistent, these static fields will be available the next time the application is launched, as expected.

Client Operations

Client Initialization

Unlike most database client libraries, our client is relatively sophisticated. Before the user's application starts, our client library intercepts the class loading process. Before loading each class, the client library examines the bytecode, performing the following functions:

1. Verify that classes meet the expected preconditions. For instance, enforcing the rules about data entities.
2. Synchronize class definitions with the database server. This provides consistency when discussing data types and method invocations.
3. Rewrite the data fields within entities. This allows lazy loading of referenced entities, thus enabling us to give the illusion that the entire object graph is in memory.
4. Add code to transparently intercept method invocations and field manipulations for entities. These operations are reported to the client library's communication manager, which notifies the server.
5. Rewrite all calls to the Java reflection API. This maintains the POJO abstractions when inspecting entities.

The client also establishes a cache to avoid round-trip calls to the database server whenever possible. When appropriate, the client may even preemptively fetch entities that are likely to be accessed.

Implementation Details

One of the most critical implementation details revolves around deciding which method invocations need to be reported to the database server. Since we used JSON (JavaScript Object Notation) and Java Serialization, both over HTTP (Hypertext Transfer Protocol), marshaling and unmarshaling are relatively expensive operations. We want to report each operation at most once, and should avoid burdening the database servers with frivolous requests. To achieve this, we use several techniques:

Call Chains

It is entirely possible for a method (caller) of one entity to call another entity method (callee). For complex data structures, these call chains are almost inevitable. If we blindly reported every method call, we would implicitly report the callee twice - once (implicitly) when we report the caller and once (explicitly) when we report the callee. To avoid this situation, we neglect to report the callee since we are already handling a method invocation. In particular, we set a thread-local variable to 'true' at the beginning of the first method in the call chain, and set the variable to 'false' at the end of that call chain. If the variable is already true, we skip the method reporting logic.

Invocation Autopromotion

Utilizing asynchronous operations allows us to avoid round trip latency to the server, but if an invocation requires a piece of uncached data, we may be unable to make forward progress without a round-trip message to the server. Under such circumstances, it makes sense to abort local execution and promote the entire call chain to a synchronous invocation, since we are already forced to pay the round-trip latency due to the cache miss. This autopromotion allows us to get a more up-to-date answer for the entire call chain at nearly zero additional cost, and allows us to avoid paying for future cache misses which could potentially occur in the same call chain when we resume local execution.

Read-only methods
It is provable that some methods will never perform write operations if those methods don't directly perform write operations, and don't invoke any methods that could perform write operations. These methods don't have any side effects worth reporting to the server, and do not need to be reported to the database server because the method invocation can be serviced completely from the local cache. Because the majority of invocations tend to be read-only, the performance savings can be substantial for both the client and the server.

NoOp methods
In practice, it is surprisingly common to see situations where a method invocation will overwrite a field with the same value, which has no effect on the entity. This is generally because the client application developer blindly calls the setter, without checking if the value has actually changed. An example might be adding an element to an EntitySet already containing that element, or blindly re-importing data from a source where most records are unchanged. In most cases, executing these methods on the primary copy would have no effect, so we may not report them to the server. In some situations, it is possible for this optimization to violate correctness when the cache is out of date, so the tradeoffs should be carefully considered. In our experience, the performance benefits seem to outweigh the potential drawbacks.

Mokanet Entity Restrictions

To ensure correctness, we place a few restrictions on classes extending the Entity class. These restrictions do not interfere with most designs, but do prohibit nonsensical operations (like trying to save an open `java.io.Socket` or running `java.util.Thread` object into the database).

Entities may only reference other entities
This rule prohibits an entity from referencing non-entities, and prevents the user from nonsensically attempting to store data types that shouldn't be saved (such as threads and sockets). The primary reason for this rule, however, is that it allows us to achieve good performance by only instrumenting objects that are going to be persisted.

If we were to permit entities to reference other objects, we would need to instrument every read, and every write, to every field within the JVM, including every array access, because any of these objects might be referenced by an entity, in which case, we would need to be aware of the access. While this isn't a technical challenge, the performance implications for applications would be substantial. By enforcing the rule that entities may only reference other entities, we instrument only a tiny fraction of the application code, and therefore have no performance impact on the rest of the application.

We find that this rule tends to encourage good design. Users must consider which data should be saved and which data is transient. It also prevents users from accidentally saving their entire JVM.

A legal exception to this rule might be to reference immutable serializable objects. Because these objects are provably immutable, they wouldn't need to be instrumented. For simplicity, we do not support this exception, though it may be permitted in the future.

Entities must be plain old Java objects (POJOs)
Entities are not permitted to perform any privileged operation, including IO, multithreading, etc. Methods should be simple and deterministic. This encourages good design by forcing the separation of the model from the control. Accessor methods should be performing simple validation and manipulations. It also allows us to invoke an entity's methods within the database server, without worrying about side effects.

Two exceptions to this rule, which we do support, are reading the current date/time (input) and printing to stdout/stderr (output). These are permitted as a convenience for logging and debugging. The output to stdout/stderr may be delayed to the conclusion of the current transaction.

Methods Signatures
Methods are permitted to accept or return any serializable data type. Nonserializable objects are not permitted in order to prevent nonsense arguments (like threads/sockets) and also to avoid the introduction of nonserializable instances into a JVM, which may violate assumptions made by the applications and thus compromise system security.

Field Visibility
As a side note, it's worth mentioning that an entity with mutable public fields is considered a warning (mokanet-lint will compilain), and is strongly discouraged for style and technical reasons. From a style perspective, accessor methods allow additional logic (such as sanity checks) to be added later without changing the public API, which is a clear win for libraries being used by third parties. There are also technical reasons why public fields are suboptimal. Although a Java class loader is capable of rewriting all Entity accesses to preserve the abstraction of public fields within a POJO entity, such an abstraction can not be maintained without analyzing the entire application. If Mokanet-based libraries are to be pre-compiled and linked in without the use of a Mokanet-aware class loader, an Entity's public fields must be made immutable (or alternatively, non-public) to avoid accidental data loss.

Performance Optimization: Lazy Reporting

Clients may choose to be lazy in their reporting of method invocations, especially in cases where such invocations are not yet visible to other clients. For example, suppose client A instantiates a new entity and performs some number of single-entity operations on this object. Client B has no way of addressing this entity, and is therefore unaware of the object's existence. Until a reference to the entity is saved to the global object graph, reporting the entity's existence to the server is unnecessary. Indeed, if client A were to crash at this point, and lose all state, the entity in question would have been garbage collected anyway, since there are no references from the root graph. Thus, correctness is not violated. Appendix A3 exemplifies this case.

The primary advantage of this technique is that some objects may be transient in nature, and never be added to the persisted object graph. In such circumstances, we are free to forget the object ever existed, thereby reducing network traffic and avoiding unnecessary server load. Another benefit is that clients may merge operations to minimize the number of reportable changes. For instance, several writes to a field can be reduced to a single write, because only the last write is relevant.

Because an unreported object is visible to only a single client, write conflicts are impossible. If clients are trustworthy, it is sufficient to keep only the entity's state, without tracking all modifications up to that point. Our system does not assume a trustworthy client, and requires an operation log to "prove" that the resulting entity is in a legal state[1]. Without an operation log, malicious clients could fabricate inconsistent or illegal entities, thereby violating assumptions made by other clients, which may have been required for correctness or even the preservation of system security guarantees. The desire to avoid the introduction of unanticipated data structures is not new, and is often cited as a key motivation for introducing the Serializable interface in the Java Language Specification.

Performance Metrics

We performed some preliminary benchmarks on the Mokanet and compared the results to equivalent operations in MySQL. When comparing these benchmarks, it's worth noting that the Mokanet numbers are still relatively unoptimized, so further performance benefits may be possible.

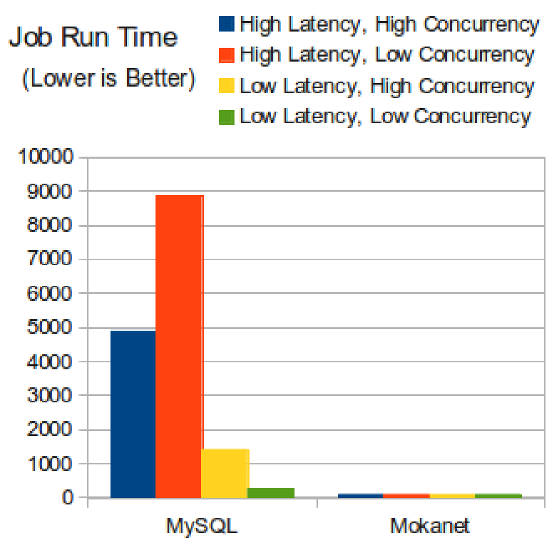
Raw reads & writes

Uncached MySQL reads and writes required an average of 26.2ms and 24.5ms respectively. Uncached Mokanet reads and writes required 22.9ms and 24.6ms respectively. Cached Mokanet reads and writes required 0.086ms 0.087ms respectively.

Concurrent Data Manipulation over High Latency Connections

Asynchronous remote method invocation really shines when an application needs to perform read-calculate-write operations. These calculations could be simple validation checks or complex mathematical computations. To exemplify this operation, we defined an integer counter field, with the integrity constraint that it always monotonically increase to the next prime number. This means that successive calls to inc() will result the counter's value containing 2, 3, 5, 7, 11, etc. To find the next value, a single transaction needs to read the current value, perform a calculation (brute-force the next prime integer), and write the new value.

We ran this experiment using both MySQL and Mokanet. In both cases, the primary copy of the counter resides on a dedicated Amazon EC2 M1.Medium instance in the US-West region, and there were 50 clients (each trying to increment the counter's value 1000 times) on an EC2 M1.Medium instance in the US-East region, to simulate high latency concurrent operations.



The Mokanet database was able to complete the task in 87.1 seconds, with zero failed/retried transactions (indicating no wasted cpu cycles by the clients or database server). The MySQL database completed the same task in 4892.7 seconds and was forced to repeat 1,111,450 transactions due to the read-write race condition. This constitutes a 56X difference.

For this benchmark, the Mokanet database was nearly insusceptible to changes in latency and/or the number of clients performing operations. MySQL, on the

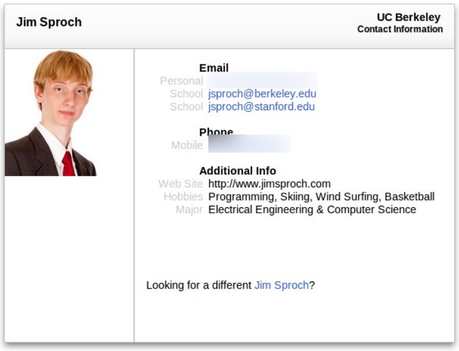
[1] A legal state is one which can be achieved through the invocation of a series of public functions. These public functions may perform checks to insure the inputs are valid. For example, the Person object described in Appendix A2 does not allow the person's birthday to be a future date (as enforced by the constructor).

other hand, was extremely susceptible to latency, and somewhat susceptible to read-write race conditions. Surprisingly, when latency is high, MySQL actually does substantially better with fifty clients than with a single client (4892.7s vs 8852.7s), despite the fact that more than two thirds of the transactions performed had to be rolled back due to read-write conflicts. By far, MySQL's best performance is realized when there is a single client (ie. no race conditions) working on a localhost database (ie. no network latency), which allows MySQL to process the workload in 249.2 seconds - but this is a totally unrealistic configuration for any large application, and is still only half as fast as the Mokanet database.

Integrated Web Service
Reads and writes were nearly all cached in our example web service (BerkeleyWho), so the runtime was approximately 250 times quicker for database calls.

BerkeleyWho

To better understand the usability of our datastore, we implemented a social networking service called BerkeleyWho for UC Berkeley students. The web application allows users to manage their contact information, manage privacy settings, and look up other students. All data is stored in the Mokanet and accessed via asynchronous remote method invocation (ARMI).



Jim Sproch

UC Berkeley
Contact Information

Email

Personal	[redacted]
School	jsproch@berkeley.edu
School	jsproch@stanford.edu

Phone

Mobile	[redacted]

Additional Info

Web Site	http://www.jimsproch.com
Hobbies	Programming, Skiing, Wind Surfing, Basketball
Major	Electrical Engineering & Computer Science

Looking for a different Jim Sproch?

Our service performs ninety-eight read operations for every two write operations, consistent with our expectation of a read-heavy work load. We were fortunate that our working set was relatively small (only ~50,000 Berkeley students & faculty) and fits on a single host, leading to a nearly 100% cache hit rate once the cache was fully populated. Cache rates and read/write ratios will, of course, vary from application to application.

Overall, we were quite satisfied with our choice to use the Mokanet and ARMI for the BerkeleyWho system. We were not bothered by the predicted results differing from the fully-consistent results, and users did not report any problems, but it's unclear if we had a sufficient number of concurrent editors to feel the full ramifications of the temporary inconsistencies.

Future Work

- Making method invocations fully transactional could eliminate the need for lock management within data structures by providing isolation. This should have negligible effects on performance, but yield substantial usability benefits by eliminating the risk of deadlocks and separating the synchronization code from application logic.
- Consistency policies in the client cache could guarantee the internal integrity of complex data structures by requiring that all entities involved in a particular operation belong to the same transactional generation. This would prevent half of a data structure being more up-to-date than the other half.
- A framework could detect and revert computations that depended on predictions which were later determined to be inaccurate, assuming that any output was delayed.

- Better static analysis could allow for further optimizations. For instance, better understanding of when new entity instances will or won't be created could allow methods (where entity instantiation is guaranteed based on the invocation arguments) to be reported asynchronously instead of synchronously.

Conclusion

Asynchronous remote method invocation (ARMI) allows us to easily communicate logical operations to be performed on arbitrarily complex data structures. The technology leverages the ubiquity of data encapsulation to provide sensible critical sections. The asynchronous communication, combined with dual execution, allows us to get reasonable predictions quickly, while maintaining strong guarantees about eventual consistency. We can all but eliminate the learning curve by utilizing class analysis and bytecode manipulation to make the persistence layer completely transparent. Because ARMI uses existing class constructs, the developer doesn't need any specialized knowledge of any persistence technologies.

References

1. Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. ACM Comput. Surv. 13, 2 (June 1981), 185-221. DOI=10.1145/356842.356846 http://doi.acm.org/10.1145/356842.356846
2. Sanny Gustavsson and Sten F. Andler. 2002. Self-stabilization and eventual consistency in replicated real-time databases. In Proceedings of the first workshop on Self-healing systems (WOSS '02), David Garlan, Jeff Kramer, and Alexander Wolf (Eds.). ACM, New York, NY, USA, 105-107. DOI=10.1145/582128.582150 http://doi.acm.org/10.1145/582128.582150
3. M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. Scads: Scale-independent storage for social computing applications. In CIDR, 2009
4. P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak, "Consistency analysis in bloom: a CALM and collected approach," in CIDR, pp. 249–260, www.crdrdb.org, 2011.
5. Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. 1997. Flexible update propagation for weakly consistent replication. In Proceedings of the sixteenth ACM symposium on Operating systems principles (SOSP '97), William M. Waite (Ed.). ACM, New York, NY, USA, 288-301. DOI=10.1145/268998.266711 http://doi.acm.org/10.1145/268998.266711
6. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," SIGOPS Oper. Syst. Rev., vol. 41, no. 6, pp. 205–220, 2007.
7. Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden. 2012. MDCC: Multi-Data Center Consistency.

Appendix A1

```
// A naive PageCounter implementation.  Notice that this class is virtually
// indistinguishable from a normal Java class.  We extend the 'Entity' class
// to indicate that this class will be persistent in the database and
// accessible by all clients.  Furthermore, it is worth noting that the
// static function increment() may be called concurrently by many clients.

public class PageCounter extends Entity
{
      public static int count;

      public static int increment()
      {
            count++;
            return count;
      }
}
```

Appendix A2

```
// Example person class, gives a clear example of a more typical data entity.
// Notice that this entity contains references to other entities (like
// the person’s phone number and email address, as well as a set of other
// Person entities (the friends).

import java.util.Date;
import com.mokanet.Entity

public class Person extends Entity
{
      String firstName;
      String lastName;
      Date birthday;
      PhoneNumber phone;
      EntitySet<Person> friends;

      public static int count;

      public Person(String firstName, string lastName, Date birthday)
      {
            this.firstName = firstName.trim();
            this.lastName = lastName.trim();

            if(birthday.getTime() > System.currentTimeInMillis())
                  throw new RuntimeExcepion("Birthday can't be in the
future");

            this.birthday = birthday;
      }

      public void addFriend(Person friend)
      {
            friends.add(friend);
      }
}

public class PhoneNumber extends Entity
{
      private String phoneNumber;

      // Perform sophisticated field validation within the data structure
      // This preserves data integrity/consistency, and fits naturally
      // into a standard Java program.
      public PhoneNumber(String phoneNumber)
      {
            try
            {
                  // Strip out characters that don't belong.  'x' is
acceptable because it represents extensions
                  phoneNumber = phoneNumber.toLowerCase().replaceAll("[^0-
9x]", "");

                  // Sometimes phone numbers are represented with a leading +1
                  if(phoneNumber.charAt(0) == '1') phoneNumber =
phoneNumber.substring(1);

                  // Perform some basic validation
                  if(!phoneNumber.matches("[0-9]{10}(x[0-9]+)?") && !
phoneNumber.matches("[0-9]11"))
                        throw new RuntimeException("Invalid phone number:
"+phoneNumber);
            }
            catch(Exception e)
            {
                  throw new ValidationException("phone number: "+phoneNumber,
e);
            }
            this.phoneNumber = phoneNumber;
      }

      // Return the phone number in a nice, readable format
      public String getFormatted()
      {
            try
            {
                  if(phoneNumber.length() == 3) return phoneNumber; // Special
case for 911, 511, 411, etc
                  int end = phoneNumber.contains("x") ?
phoneNumber.indexOf('x') : phoneNumber.length();
                  return "("+phoneNumber.substring(0, 3)+")
"+phoneNumber.substring(3, 6)+"-"+phoneNumber.substring(6, end)+
(phoneNumber.contains("x") ? " (x"+phoneNumber.substring(end+1)+")" : "");

            }
            catch(Exception e)
            {
                  throw new ValidationException("invalid phone number:
"+phoneNumber, e);
            }
      }
}
```

Appendix A3

```
// For the purpose of this example, we assume Book is a data entity class
// with the appropriate method defined.  Note that we have created a new book,
// but there is no way for another client to get a reference to this book,
// so we don’t need to report any of these invocations (we can be lazy).

Example operations can be lazy reported
Book b = new Book(“Harry Potter”);
b.setAuthor(Author.lookup(“J. K. Rowling”));
b.setISBN(“978-0590353427”);
b.setPageCount(320);
b.setPublisher(“Scholastic Paperbacks”);

// If we save the book to someone’s favorites list (assuming the person is
// already saved in the database), we need to flush our queue of changes to
the
// book, because the book is now part of the database graph (accessible from
// another part of the database and therefore accessible by another db
client).

person.addToFavorites(b);
```