Hammersmith: Netty, Scala, and MongoDB

Brendan W. McAdams

10gen, Inc.

NY Scala Enthusiasts

What is Hammersmith?

Overview

- New MongoDB Driver for Scala
- Distillation of Lessons Learned in 18 Months of Casbah (Spiritual "cousin" rather than successor)
- Pure Scala (With a little bit of Java at the very bottom for BSON)
- Focused more on *frameworks* than *userspace*
- Purely Asynchronous and Non-Blocking Network I/O

mongoDB

What is Hammersmith?

What Problems Does It Solve/Explore?

- Learn BSON and the MongoDB Wire Protocol
- Architectural Evolutions from Java Driver (and Casbah Lessons)
  - Better Connection Pools and Cleaner Type tree for Connections (Direct, Replica Set, Master/Slave)
  - Faster, cleaner and more extensible pluggability for custom serialization of business objects <-> BSON
- Asynchronous Networking
  - Integrate more appropriately with purely asynchronous frameworks like "BlueEyes" `https://github.com/jdegoes/blueeyes`
  - Get away from any synchronization, threading and blocking which can limit the scalability ceiling of working with MongoDB
- ~~World Domination!~~

Speaking to MongoDB Asynchronously I

- In an asynchronous framework, we must be careful to never block
- The problem of course, is how do you manage a truly synchronous operation like talking to a database?

Speaking to MongoDB Asynchronously II

- Anonymous Function Callbacks and Dispatching by Request ID does the trick nicely!

```
/**
 * The request ID and callback will be stored in a ConcurrentMap when the write is
     sent and
 * invoked when a reply for that request comes back
 */
def databaseNames(callback: Seq[String] => Unit) {
  runCommand("admin", Document("listDatabases" ->
      1))(SimpleRequestFutures.command((doc: Document) => {
    log.debug("Got a result from 'listDatabases' command: %s", doc)
    if (!doc.isEmpty) {
      val dbs =
      doc.as[BSONList]("databases").asList.map(_.asInstanceOf[Document].as[String]("name"))
      callback(dbs)
    } else {
      log.warning("Command 'listDatabases' failed. Doc: %s", doc)
      callback(List.empty[String])
    }
  }))
}

/**
 * SimpleRequestFutures "swallows" any exceptions, as many times people want to ignore
     them.
 * For those who want to handle any error by hand, the underlying code uses
     Either[Throwable, A].
 */
```

Speaking to MongoDB Asynchronously III

```
command(authCmd)(RequestFutures.findOne((result: Either[Throwable, Document]) => {
  result match {
    case Right(_doc) =>
      _doc.getAsOrElse[Int]("ok", 0) match {
        case 1 => {
          log.info("Authenticate succeeded.")
          login = Some(username)
          authHash = Some(hash)
        }
        case other => log.error("Authentication Failed. '%d' OK status. %s", other,
      _doc)
      }
    case Left(e) =>
      log.error(e, "Authentication Failed.")
      callback(this)
  }
}))

// A base trait for RequestFutures which handles the application.
sealed trait RequestFuture {
  type T
  val body: Either[Throwable, T] => Unit

  def apply(error: Throwable) = body(Left(error))

  def apply[A <% T](result: A) = body(Right(result.asInstanceOf[T]))
}

trait CursorQueryRequestFuture extends RequestFuture {
  type T <: Cursor
```

Speaking to MongoDB Asynchronously IV

```
}

/**
 *
 * Used for findOne and commands
 * Items which return a single document, and not a cursor
 */
trait SingleDocQueryRequestFuture extends QueryRequestFuture {
  type T <: BSONDocument
}

// Finally, the helper methods just provide convenience
// "Simple"
def command[A <: BSONDocument](f: A => Unit) =
  new SingleDocQueryRequestFuture {
    type T = A
    val body = (result: Either[Throwable, A]) => result match {
      case Right(doc) => f(doc)
      case Left(t) => log.error(t, "Command Failed.")
    }
  }

// "Handle your own damn errors"
def command[A <: BSONDocument](f: Either[Throwable, A] => Unit) =
  new SingleDocQueryRequestFuture {
    type T = A
    val body = f
  }
```

Speaking to MongoDB Asynchronously I

Cursors

- Handling a findOne or command, which return a single Document is not difficult.
- Because of batching, Cursors (Which MongoDB uses where number of matching docs > 1) are difficult to do asynchronously without blocking
- Standard iteration (say, calling "next()") has in essence, two return values:
  - Some(value) ... A valid value was found in the iterator and returned
  - None ... No value was found, this iterator is empty and done

Speaking to MongoDB Asynchronously II

Cursors

- A cursor (MongoDB's are much like any other database's) returns documents in batches to save memory on the client. They really have *three* states:
  - Entry(value) ... A valid value was found in the *CLIENTS LOCAL BATCH* and returned
  - Empty ... The client's local batch is empty but there appear to be more results on the server
  - EOF ... The client and server have exhausted their results and nothing more is to be had.
- Solution: Haskell's Iteratee Pattern (also available in scalaz). Suggested by @jdegoes (thanks!)
- The three states are represented as an "IterState" and the cursor iteration is controlled by "IterCmd" issued by a user function in response to "IterState".

Speaking to MongoDB Asynchronously III

Cursors

```
log.debug("[%s] Querying for Collection Names with: %s", name, qMsg)
connection.send(qMsg, SimpleRequestFutures.find((cursor: Cursor) => {
  log.debug("Got a result from listing collections: %s", cursor)
  val b = Seq.newBuilder[String]

  Cursor.basicIter(cursor) { doc =>
    val n = doc.as[String]("name")
    if (!n.contains("$")) b += n.split(name + "\\.")(1)
  }

  callback(b.result())
}))

// The function helpers for iteration ...

protected[mongodb] def basicIter(cursor: Cursor)(f: BSONDocument => Unit) = {
  def next(op: Cursor.IterState): Cursor.IterCmd = op match {
    case Cursor.Entry(doc) =>  {
      f(doc)
      Cursor.Next(next)
    }
    case Cursor.Empty => {
      Cursor.NextBatch(next)
    }
    case Cursor.EOF => {
      Cursor.Done
    }
  }
```

Speaking to MongoDB Asynchronously IV

Cursors

```
  iterate(cursor)(next)
}

def iterate(cursor: Cursor)(op: (IterState) => IterCmd) {
  log.debug("Iterating '%s' with op: '%s'", cursor, op)
  @tailrec def next(f: (IterState) => IterCmd): Unit = op(cursor.next()) match {
    case Done => {
      log.info("Closing Cursor.")
      cursor.close()
    }
    case Next(tOp) => {
      log.debug("Next!")
      next(tOp)
    }
    case NextBatch(tOp) => cursor.nextBatch(() => {
      log.info("Next Batch Loaded.")
      next(tOp)
    })
  }
  next(op)
}
```

What's Next? I

- Still fleshing out the BSON layer, but focused on being able to plug completely custom Business Objects in at a low level
- Working on Connection Pools (with commons-pool)
- Aiming to Release an Alpha in the next 2-3 weeks
- The goal here is to be partly community driven: Making this fit into the frameworks in a way that benefits the Scala + MongoDB user bases. I want your contributions and thoughts!

Questions?

- Twitter: **@rit** | mongodb: **@mongodb** | 10gen: **@10gen**
- email: **brendan@10gen.com**
- Upcoming MongoDB Events
  - Mongo Philly (April 26, 2011) - `http://bit.ly/mongophilly`
  - Mongo NY (June 7, 2011) - `http://www.10gen.com/conferences/mongonyc2011`
- 10gen is hiring! We need smart engineers (C++) in both NY and the Bay Area (Redwood Shores): `http://10gen.com/jobs`

mongoDB