Searchbird «Previous

We're going to build a simple distributed search engine using Scala and the previously discussed Finagle framework.

Design goals: the big picture

Broadly, our design goals include *abstraction* (the ability to use the resulting system without knowing all of its internal detail); *modularity* (the ability to factor the system into smaller, simpler pieces that can be more easily understood and/or replaced with other pieces); and *scalability* (the ability to grow the capacity of the system in a straightforward way).

The system we're going to describe has three pieces: (1) *clients* that makes requests to (2) *servers*, which send responses to the request; and a (3) *transport* mechanism that packages up these communications. Normally the client and server would be located on different machines and communicate over a network on a particular numerical *port*, but in this example, they will run on the same machine (and still communicate using ports). In our example, clients and servers will be written in Scala, and the transport will be handled using Thrift. The primary purpose of this tutorial is to show a simple server and client that can be extended to provide scalable performance.

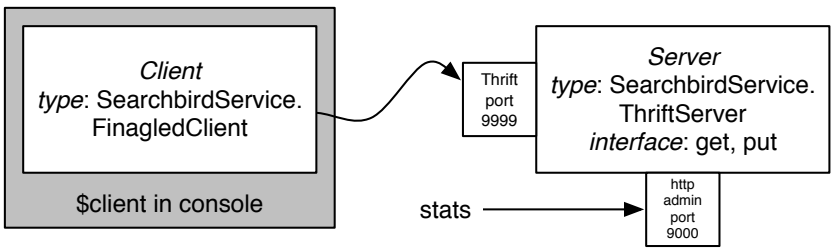
Exploring the default bootstrapper project

First, create a skeleton project ("Searchbird") using scala-bootstrapper. This creates a simple Finagle -based Scala service that exports an in-memory key-value store. We'll extend this to support searching of the values, and then extend it to support searching multiple in-memory stores over several processes.

```
$ mkdir searchbird ; cd searchbird
$ scala-bootstrapper searchbird
writing build.sbt
writing config/development.scala
writing config/production.scala
writing config/staging.scala
writing config/test.scala
writing console
writing Gemfile
writing project/plugins.sbt
writing README.md
writing sbt
writing src/main/scala/com/twitter/searchbird/SearchbirdConsoleClient.scala
writing src/main/scala/com/twitter/searchbird/SearchbirdServiceImpl.scala
writing src/main/scala/com/twitter/searchbird/config/SearchbirdServiceConfig.scala
writing src/main/scala/com/twitter/searchbird/Main.scala
writing src/main/thrift/searchbird.thrift
writing src/scripts/searchbird.sh
writing src/scripts/config.sh
writing src/scripts/devel.sh
writing src/scripts/server.sh
writing src/scripts/service.sh
writing src/test/scala/com/twitter/searchbird/AbstractSpec.scala
writing src/test/scala/com/twitter/searchbird/SearchbirdServiceSpec.scala
writing TUTORIAL.md
```

Let's first explore the default project `scala-bootstrapper` creates for us. This is meant as a template. You'll end up substituting most of it, but it serves as a convenient scaffold. It defines a simple (but complete) key-value store. Configuration, a thrift interface, stats export and logging are all included.

Before we look at code, we're going to run a client and server to see how it works. Here's what we're building:



and here is the interface that our service exports. Since the Searchbird service is a Thrift service (like most of our services), its external interface is defined in the Thrift IDL ("interface description language").

src/main/thrift/searchbird.thrift

```
service SearchbirdService {
  string get(1: string key) throws(1: SearchbirdException ex)

  void put(1: string key, 2: string value)
}
```

This is pretty straightforward: our service `SearchbirdService` exports 2 RPC methods, `get` and `put`. They comprise a simple interface to a key-value store.

Now let's run the default service and a client that connects to this service, and explore them through this interface. Open two windows, one for the server and one for the client.

In the first window, start sbt in interactive mode (run `./sbt` from the command line[1]) and then build and run the project from within sbt. This runs the `main` routine in `Main.scala`.

```
$ ./sbt
...
> compile
> run -f config/development.scala
...
[info] Running com.twitter.searchbird.Main -f config/development.scala
```

The configuration file (`development.scala`) instantiates a new service, exposing that service on port 9999 on our local machine. Clients can communicate with this service by connecting to port 9999.

Now we're going to instantiate and run a client using the provided `console` shell script, which instantiates a `SearchbirdConsoleClient` instance (from `SearchbirdConsoleClient.scala`). Run this script in the other window:

```
$ ./console 127.0.0.1 9999
[info] Running com.twitter.searchbird.SearchbirdConsoleClient 127.0.0.1 9999
'client' is bound to your thrift client.

finagle-client>
```

The client object `client` is now connected to port 9999 on our local computer, and can talk to the service that we previously started on that port. So let's send it some requests:

```
scala> client.put("marius", "Marius Eriksen")
res0: ...

scala> client.put("stevej", "Steve Jenson")
res1: ...

scala> client.get("marius")
res2: com.twitter.util.Future[String] = ...

scala> client.get("marius").get()
res3: String = Marius Eriksen
```

(The second `get()` call resolves the `Future` that is the return type of `client.get()` by blocking until the value is ready.)

The server also exports runtime statistics (the configuration file specifies these can be found at port 9900). These are convenient both for inspecting individual servers as well as aggregating into global service statistics (a machine-readable JSON interface is also provided). Open a third window and check those stats:

```
$ curl localhost:9900/stats.txt
counters:
  Searchbird/connects: 1
  Searchbird/received_bytes: 264
  Searchbird/requests: 3
  Searchbird/sent_bytes: 128
  Searchbird/success: 3
  jvm_gc_ConcurrentMarkSweep_cycles: 1
  jvm_gc_ConcurrentMarkSweep_msec: 15
  jvm_gc_ParNew_cycles: 24
  jvm_gc_ParNew_msec: 191
  jvm_gc_cycles: 25
  jvm_gc_msec: 206
gauges:
  Searchbird/connections: 1
  Searchbird/pending: 0
  jvm_fd_count: 135
  jvm_fd_limit: 10240
  jvm_heap_committed: 85000192
  jvm_heap_max: 530186240
  jvm_heap_used: 54778640
  jvm_nonheap_committed: 89657344
  jvm_nonheap_max: 136314880
  jvm_nonheap_used: 66238144
```

```
  jvm_num_cpus: 4
  jvm_post_gc_CMS_Old_Gen_used: 36490088
  jvm_post_gc_CMS_Perm_Gen_used: 54718880
  jvm_post_gc_Par_Eden_Space_used: 0
  jvm_post_gc_Par_Survivor_Space_used: 1315280
  jvm_post_gc_used: 92524248
  jvm_start_time: 1345072684280
  jvm_thread_count: 16
  jvm_thread_daemon_count: 7
  jvm_thread_peak_count: 16
  jvm_uptime: 1671792
labels:
metrics:
  Searchbird/handletime_us: (average=9598, count=4, maximum=19138, minimum=637, p25=637, p50=4265, p75=14175, p90=19138,
p95=19138, p99=19138, p999=19138, p9999=19138, sum=38393)
  Searchbird/request_latency_ms: (average=4, count=3, maximum=9, minimum=0, p25=0, p50=5, p75=9, p90=9, p95=9, p99=9, p999=9,
p9999=9, sum=14)
```

In addition to our own service statistics, we are also given some generic JVM stats that are often useful.

Now let's look at the code that implements the configuration, the server, and the client.

.../config/SearchbirdServiceConfig.scala

A configuration is a Scala trait that has a method `apply: RuntimeEnvironment => T` for some `T` we want to create. In this sense, configurations are "factories". At runtime, a configuration file is evaluated as a script (by using the scala compiler as a library), and is expected to produce such a configuration object. A `RuntimeEnvironment` is an object queried for various runtime parameters (command-line flags, JVM version, build timestamps, etc.).

The `SearchbirdServiceConfig` class specifies such a class. It specifies configuration parameters together with their defaults. (Finagle supports a generic tracing system that we can ignore for the purposes of this tutorial; the Zipkin distributed tracing system is a collector/aggregator of such traces.)

```
class SearchbirdServiceConfig extends ServerConfig[SearchbirdService.ThriftServer] {
  var thriftPort: Int = 9999
  var tracerFactory: Tracer.Factory = NullTracer.factory

  def apply(runtime: RuntimeEnvironment) = new SearchbirdServiceImpl(this)
}
```

In our case, we want to create a `SearchbirdService.ThriftServer`. This is the server type generated by the thrift code generator[2].

.../Main.scala

Typing "run" in the sbt console calls `main`, which configures and instantiates this server. It reads the configuration (specified in `development.scala` and supplied as an argument to "run"), creates the `SearchbirdService.ThriftServer`, and starts it. `RuntimeEnvironment.loadRuntimeConfig` performs the configuration evaluation and calls its `apply` method with itself as an argument[3].

```
object Main {
  private val log = Logger.get(getClass)

  def main(args: Array[String]) {
    val runtime = RuntimeEnvironment(this, args)
    val server = runtime.loadRuntimeConfig[SearchbirdService.ThriftServer]
    try {
      log.info("Starting SearchbirdService")
      server.start()
    } catch {
      case e: Exception =>
        log.error(e, "Failed starting SearchbirdService, exiting")
        ServiceTracker.shutdown()
        System.exit(1)
    }
  }
}
```

.../SearchbirdServiceImpl.scala

This is the meat of the service: we extend the `SearchbirdService.ThriftServer` with our custom implementation. Recall that `SearchbirdService.ThriftServer` has been created for us by the thrift code generator. It generates a scala method per thrift method. In our example so far, the generated interface is:

```
trait SearchbirdService {
  def put(key: String, value: String): Future[Void]
  def get(key: String): Future[String]
```

```
}
```

`Future[Value]`s are returned instead of the values directly so that their computation may be deferred (finagle's documentation has more details on `Future`). For the purpose of this tutorial, the only thing you need to know about a `Future` is that you can retrieve its value with `get()`.

The default implementation of the key-value store provided by `scala-bootstrapper` is straightforward: it provides a `database` data structure and `get` and `put` calls that access that data structure.

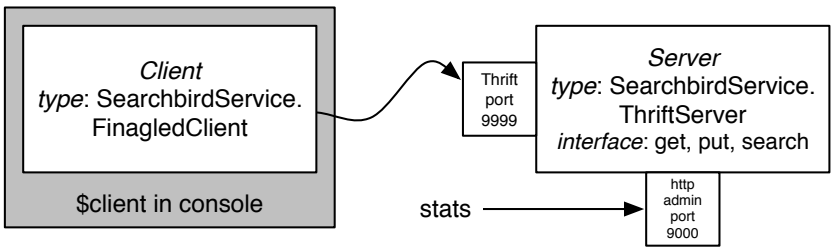
```
class SearchbirdServiceImpl(config: SearchbirdServiceConfig) extends SearchbirdService.ThriftServer {
  val serverName = "Searchbird"
  val thriftPort = config.thriftPort
  override val tracerFactory = config.tracerFactory

  val database = new mutable.HashMap[String, String]()

  def get(key: String) = {
    database.get(key) match {
      case None =>
        log.debug("get %s: miss", key)
        Future.exception(SearchbirdException("No such key"))
      case Some(value) =>
        log.debug("get %s: hit", key)
        Future(value)
    }
  }

  def put(key: String, value: String) = {
    log.debug("put %s", key)
    database(key) = value
    Future.Unit
  }

  def shutdown() = {
    super.shutdown(0.seconds)
  }
}
```

The result is a simple thrift interface to a Scala `HashMap`.

A simple search engine

Now we'll extend our example so far to create a simple search engine. We'll then extend it further to become a *distributed* search engine consisting of multiple shards so that we can fit a corpus larger than what can fit in the memory of a single machine.

To keep things simple, we'll extend our current thrift service minimally in order to support a search operation. The usage model is to `put` documents into the search engine, where each document contains a series of tokens (words); then we can search on a string of tokens to return all documents that contain all tokens in the set. The architecture is identical to the previous example but for the addition of a new `search` call.



To implement such a search engine, make the following changes to the following two files:

src/main/thrift/searchbird.thrift

```
service SearchbirdService {
  string get(1: string key) throws(1: SearchbirdException ex)

  void put(1: string key, 2: string value)

  list<string> search(1: string query)
}
```

We've added a `search` method that searches the current hashtable, returning the list of keys whose values match the query. The implementation is also straightforward:

.../SearchbirdServiceImpl.scala

Most of our changes take place in this file.

The current `database` hashmap holds a forward index that maps a key to a document. We rename it to `forward` and add a second map for the `reverse` index (that maps a token to the set of documents that contain that token). So, within `SearchbirdServiceImpl.scala`, replace the `database` definition with:

```
val forward = new mutable.HashMap[String, String]
  with mutable.SynchronizedMap[String, String]
val reverse = new mutable.HashMap[String, Set[String]]
  with mutable.SynchronizedMap[String, Set[String]]
```

Within the `get` call, replace `database` with `forward`, but otherwise, `get` remains the same (it only performs forward lookups). However, `put` requires changes: we also need to populate the reverse index for each token in the document by appending the document key to the list associated with that token. Replace the `put` call with the following code. Given a particular search token, we can now use the `reverse` map to look up documents.

```
def put(key: String, value: String) = {
  log.debug("put %s", key)

  forward(key) = value

  // serialize updaters
  synchronized {
    value.split(" ").toSet foreach { token =>
      val current = reverse.getOrElse(token, Set())
      reverse(token) = current + key
    }
  }

  Future.Unit
}
```

Note that (even though the `HashMap` is thread-safe) only one thread can update the `reverse` map at a time to ensure that read-modify-write of a particular map entry is an atomic operation. (The code is overly conservative; it locks the entire map rather than locking each individual retrieve-modify-write operation.) Also note the use of `Set` as the data structure; this ensures that if the same token appears twice in a document, it will only be processed by the `foreach` loop once.

The implementation still has an issue that is left as an exercise for the reader: when we overwrite a key with a new document, we don't remove any references to the old document in the reverse index.

Now to the meat of the search engine: the new `search` method. It should tokenize its query, look up all of the matching documents and, then intersect these lists. This will yield the list of documents that contain all of the tokens in the query. This is straightforward to express in Scala; add this to the `SearchbirdServiceImpl` class:

```
def search(query: String) = Future.value {
  val tokens = query.split(" ")
  val hits = tokens map { token => reverse.getOrElse(token, Set()) }
  val intersected = hits reduceLeftOption { _ & _ } getOrElse Set()
  intersected.toList
}
```

A few things are worth calling out in this short piece of code. When constructing the hit list, if the key (`token`) is not found, `getOrElse` will supply the value from its second parameter (in this case, an empty `Set`). We perform the actual intersection using a left-reduce. The particular flavor, `reduceLeftOption`, will not attempt to perform the reduce if `hits` is empty, returning instead `None`. This allows us to supply a default value instead of experiencing an exception. In fact, this is equivalent to:

```
def search(query: String) = Future.value {
  val tokens = query.split(" ")
  val hits = tokens map { token => reverse.getOrElse(token, Set()) }
  if (hits.isEmpty)
    Nil
  else
    hits reduceLeft { _ & _ } toList
}
```

Which to use is mostly a matter of taste, though functional style often eschews conditionals for sensible defaults.

We can now experiment with our new implementation using the console. Start your server again:

```
$ ./sbt
```

```
...
> compile
> run -f config/development.scala
...
[info] Running com.twitter.searchbird.Main -f config/development.scala
```

and then from the searchbird directory, start up a client:

```
$ ./console 127.0.0.1 9999
...
[info] Running com.twitter.searchbird.SearchbirdConsoleClient 127.0.0.1 9999
'client' is bound to your thrift client.

finagle-client>
```

Paste the following lecture descriptions into the console:

```
client.put("basics", " values functions classes methods inheritance try catch finally expression oriented")
client.put("basics", " case classes objects packages apply update functions are objects (uniform access principle) pattern")
client.put("collections", " lists maps functional combinators (map foreach filter zip")
client.put("pattern", " more functions! partialfunctions more pattern")
client.put("type", " basic types and type polymorphism type inference variance bounds")
client.put("advanced", " advanced types view bounds higher kinded types recursive types structural")
client.put("simple", " all about sbt the standard scala build")
client.put("more", " tour of the scala collections")
client.put("testing", " write tests with specs a bdd testing framework for")
client.put("concurrency", " runnable callable threads futures twitter")
client.put("java", " java interop using scala from")
client.put("searchbird", " building a distributed search engine using")
```

We can now perform some searches, which return the keys of the documents that contain the search terms.

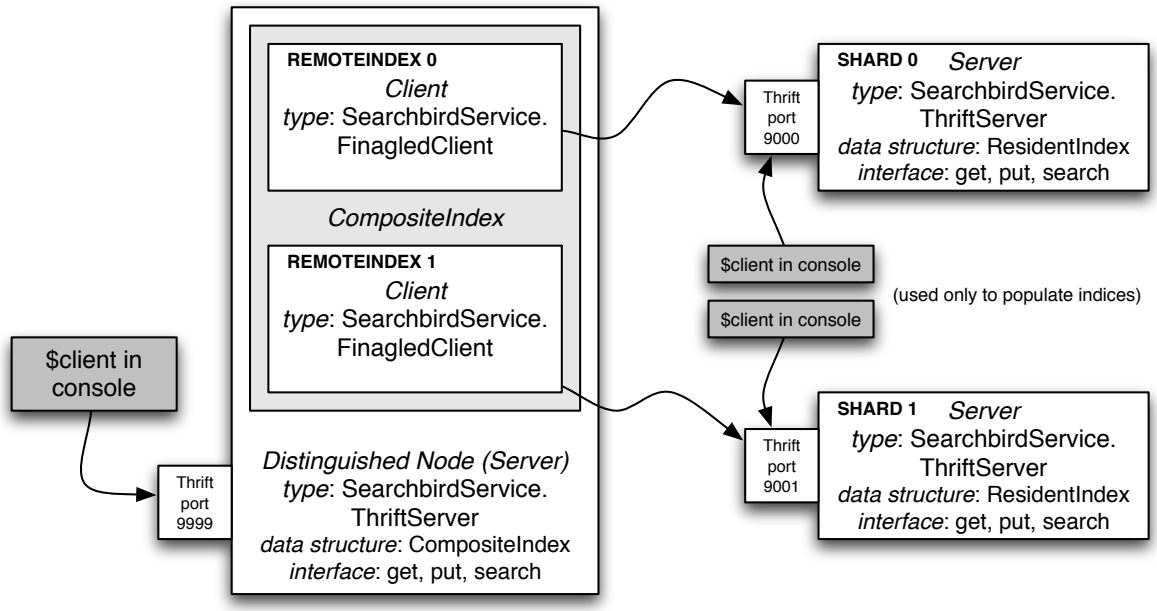
```
> client.search("functions").get()
res12: Seq[String] = ArrayBuffer(basics)

> client.search("java").get()
res13: Seq[String] = ArrayBuffer(java)

> client.search("java scala").get()
res14: Seq[String] = ArrayBuffer(java)

> client.search("functional").get()
res15: Seq[String] = ArrayBuffer(collections)

> client.search("sbt").get()
res16: Seq[String] = ArrayBuffer(simple)

> client.search("types").get()
res17: Seq[String] = ArrayBuffer(type, advanced)
```

Recall that if the call returns a `Future`, we have to use a blocking `get()` call to resolve the value contained within that future. We can use the `Future.collect` command to make multiple concurrent requests and wait for all of them to succeed:

```
> import com.twitter.util.Future
...
> Future.collect(Seq(
    client.search("types"),
    client.search("sbt"),
    client.search("functional")
  )).get()
res18: Seq[Seq[String]] = ArrayBuffer(ArrayBuffer(type, advanced), ArrayBuffer(simple), ArrayBuffer(collections))
```

Distributing our service

On a single machine, our simple in-memory search engine won't be able to search a corpus larger than the size of memory. We'll now venture to remedy this by distributing nodes with a simple sharding scheme. Here's the block diagram:



Abstracting

To aid our work, we'll first introduce another abstraction—an `Index`—in order to decouple the index implementation from the `SearchbirdService`. This is a straightforward refactor. We'll begin by adding an Index file to the build (create the file `searchbird/src/main/scala/com/twitter/searchbird/Index.scala`):

.../Index.scala

```
package com.twitter.searchbird

import scala.collection.mutable
import com.twitter.util._
import com.twitter.conversions.time._
import com.twitter.logging.Logger
import com.twitter.finagle.builder.ClientBuilder
import com.twitter.finagle.thrift.ThriftClientFramedCodec

trait Index {
  def get(key: String): Future[String]
  def put(key: String, value: String): Future[Unit]
  def search(key: String): Future[List[String]]
}

class ResidentIndex extends Index {
  val log = Logger.get(getClass)

  val forward = new mutable.HashMap[String, String]
    with mutable.SynchronizedMap[String, String]
  val reverse = new mutable.HashMap[String, Set[String]]
    with mutable.SynchronizedMap[String, Set[String]]

  def get(key: String) = {
    forward.get(key) match {
      case None =>
        log.debug("get %s: miss", key)
        Future.exception(SearchbirdException("No such key"))
      case Some(value) =>
        log.debug("get %s: hit", key)
        Future(value)
    }
  }

  def put(key: String, value: String) = {
    log.debug("put %s", key)

    forward(key) = value

    // admit only one updater.
    synchronized {
      (Set() ++ value.split(" ")) foreach { token =>
```

```
        val current = reverse.get(token) getOrElse Set()
        reverse(token) = current + key
      }
    }

    Future.Unit
  }

  def search(query: String) = Future.value {
    val tokens = query.split(" ")
    val hits = tokens map { token => reverse.getOrElse(token, Set()) }
    val intersected = hits reduceLeftOption { _ & _ } getOrElse Set()
    intersected.toList
  }
}
```

We now convert our thrift service to a simple dispatch mechanism: it provides a thrift interface to any `Index` instance. This is a powerful abstraction, because it separates the implementation of the service from the implementation of the index. The service no longer has to know any details of the underlying index; the index might be local or might be remote or might be a composite of many remote indices, but the service doesn't care, and the implementation of the index might change without the service changing.

Replace your `SearchbirdServiceImpl` class definition with the (much simpler) one below (which no longer contains the index implementation detail). Note initializing a server now takes a second argument, an `Index`.

.../SearchbirdServiceImpl.scala

```
class SearchbirdServiceImpl(config: SearchbirdServiceConfig, index: Index) extends SearchbirdService.ThriftServer {
  val serverName = "Searchbird"
  val thriftPort = config.thriftPort

  def get(key: String) = index.get(key)
  def put(key: String, value: String) =
    index.put(key, value) flatMap { _ => Future.Unit }
  def search(query: String) = index.search(query)

  def shutdown() = {
    super.shutdown(0.seconds)
  }
}
```

.../config/SearchbirdServiceConfig.scala

Update your `apply` call in `SearchbirdServiceConfig` accordingly:

```
class SearchbirdServiceConfig extends ServerConfig[SearchbirdService.ThriftServer] {
  var thriftPort: Int = 9999
  var tracerFactory: Tracer.Factory = NullTracer.factory

  def apply(runtime: RuntimeEnvironment) = new SearchbirdServiceImpl(this, new ResidentIndex)
}
```

We'll set up our simple distributed system so that there is one distinguished node that coordinates queries to its child nodes. In order to achieve this, we'll need two new `Index` types. One represents a remote index, the other is a composite index over several other `Index` instances. This way we can construct the distributed index by instantiating a composite index of the remote indices. Note that both `Index` types have the same interface, so servers do not need to know whether the index to which they are connected is remote or composite.

.../Index.scala

In `Index.scala`, define a `CompositeIndex`:

```
class CompositeIndex(indices: Seq[Index]) extends Index {
  require(!indices.isEmpty)

  def get(key: String) = {
    val queries = indices.map { idx =>
      idx.get(key) map { r => Some(r) } handle { case e => None }
    }

    Future.collect(queries) flatMap { results =>
      results.find { _.isDefined } map { _.get } match {
        case Some(v) => Future.value(v)
        case None => Future.exception(SearchbirdException("No such key"))
```

```
      }
    }
  }

  def put(key: String, value: String) =
    Future.exception(SearchbirdException("put() not supported by CompositeIndex"))

  def search(query: String) = {
    val queries = indices.map { _.search(query) rescue { case _=> Future.value(Nil) } }
    Future.collect(queries) map { results => (Set() ++ results.flatten) toList }
  }
}
```

The composite index works over a set of underlying `Index` instances. Note that it doesn't care how these are actually implemented. This type of composition allows for great flexibility in constructing various querying schemes. We don't define a sharding scheme, and so the composite index doesn't support `put` operations. These are instead issued directly to the child nodes. `get` is implemented by querying all of our child nodes and picking the first successful result. If there are none, we throw an exception. Note that since the absence of a value is communicated by throwing an exception, we `handle` this on the `Future`, converting any exception into a `None` value. In a real system, we'd probably have proper error codes for missing values rather than using exceptions. Exceptions are convenient and expedient for prototyping, but compose poorly. In order to distinguish between a real exception and a missing value, I have to examine the exception itself. Rather, it is better style to embed this distinction directly in the type of the returned value.

`search` works in a similar way as before. Instead of picking the first result, we combine them, ensuring their uniqueness by using a `Set` construction.

`RemoteIndex` provides an `Index` interface to a remote server.

```
class RemoteIndex(hosts: String) extends Index {
  val transport = ClientBuilder()
    .name("remoteIndex")
    .hosts(hosts)
    .codec(ThriftClientFramedCodec())
    .hostConnectionLimit(1)
    .timeout(500.milliseconds)
    .build()
  val client = new SearchbirdService.FinagledClient(transport)

  def get(key: String) = client.get(key)
  def put(key: String, value: String) = client.put(key, value) map { _ => () }
  def search(query: String) = client.search(query) map { _.toList }
}
```

This constructs a finagle thrift client with some sensible defaults, and just proxies the calls, adjusting the types slightly.

Putting it all together

We now have all the pieces we need. We'll need to adjust the configuration in order to be able to invoke a given node as either a distinguished node or a data shard node. In order to do so, we'll enumerate the shards in our system by creating a new config item for it. We also need to add the `Index` argument to our instantiation of the `SearchbirdServiceImpl`. We'll then use command line arguments (recall that the `Config` has access to these) to start the server up in either mode.

.../config/SearchbirdServiceConfig.scala

```
class SearchbirdServiceConfig extends ServerConfig[SearchbirdService.ThriftServer] {
  var thriftPort: Int = 9999
  var shards: Seq[String] = Seq()

  def apply(runtime: RuntimeEnvironment) = {
    val index = runtime.arguments.get("shard") match {
      case Some(arg) =>
        val which = arg.toInt
        if (which >= shards.size || which < 0)
          throw new Exception("invalid shard number %d".format(which))

        // override with the shard port
        val Array(_, port) = shards(which).split(":")
        thriftPort = port.toInt

        new ResidentIndex

      case None =>
        require(!shards.isEmpty)
        val remotes = shards map { new RemoteIndex(_) }
        new CompositeIndex(remotes)
    }
```

```
    new SearchbirdServiceImpl(this, index)
  }
}
```

Now we'll adjust the configuration itself: add the "shards" initialization to the instantiation of `SearchbirdServiceConfig` (we can talk to shard 0 via port 9000, shard 1 via port 9001, and so on).

config/development.scala

```
new SearchbirdServiceConfig {
  // Add your own config here
  shards = Seq(
    "localhost:9000",
    "localhost:9001",
    "localhost:9002"
  )
  ...
```

Comment out the setting for `admin.httpPort` (we don't want several services running on the same machine, all of which are trying to open up the same port):

```
  // admin.httpPort = 9900
```

Now if we invoke our server without any arguments, it starts a distinguished node that speaks to all of the given shards. If we specify a shard argument, it starts a server on the port belonging to that shard index.

Let's try it! We'll launch 3 services: 2 shards and 1 distinguished node. First compile the changes:

```
$ ./sbt
> compile
...
> exit
```

Then launch 3 servers:

```
$ ./sbt 'run -f config/development.scala -D shard=0'
$ ./sbt 'run -f config/development.scala -D shard=1'
$ ./sbt 'run -f config/development.scala'
```

You can either run these in 3 different windows or (in the same window) start each one in turn, wait for it to start up, control-z to suspend it, and `bg` to put it running in the background.

Then we'll interact with them through the console. First, let's populate some data in the two shard nodes. Running from the searchbird directory:

```
$ ./console localhost 9000
...
> client.put("fromShardA", "a value from SHARD_A")
> client.put("hello", "world")
```

```
$ ./console localhost 9001
...
> client.put("fromShardB", "a value from SHARD_B")
> client.put("hello", "world again")
```

You can exit these console sessions once you complete them. Now query our database from the distinguished node (port 9999):

```
$ ./console localhost 9999
[info] Running com.twitter.searchbird.SearchbirdConsoleClient localhost 9999
'client' is bound to your thrift client.

finagle-client> client.get("hello").get()
res0: String = world

finagle-client> client.get("fromShardC").get()
SearchbirdException(No such key)
...
```

```
finagle-client> client.get("fromShardA").get()
res2: String = a value from SHARD_A

finagle-client> client.search("hello").get()
res3: Seq[String] = ArrayBuffer()

finagle-client> client.search("world").get()
res4: Seq[String] = ArrayBuffer(hello)

finagle-client> client.search("value").get()
res5: Seq[String] = ArrayBuffer(fromShardA, fromShardB)
```

This design has multiple data abstractions that allow a more modular and scalable implementation:

- The `ResidentIndex` data structure knows nothing about the network, servers, or clients.
- The `CompositeIndex` knows nothing about how its constituent indices are implemented or their underlying data structures; it simply distributes its requests to them.
- The same `search` interface (trait) for servers allows a server to query its local data structure (`ResidentIndex`) or distribute queries to other servers (`CompositeIndex`) without needing to know this distinction, which is hidden from the caller.
- The `SearchbirdServiceImpl` and `Index` are separate modules now, allowing a simple implementation of the service and separating the implementation of the data structure from the service that accesses it.
- The design is flexible enough to allow one or many remote indices, located on the local machine or on remote machines.

Possible improvements to this implementation would include:

- The current implementation sends `put()` calls to all nodes. Instead, we could use a hash table to send a `put()` call to only one node and distribute storage across all nodes.
  - Note, however, we lose redundancy with this strategy. How could we maintain some redundancy yet not require full replication?
- We aren't doing anything interesting with any failures in the system (we aren't processing any exceptions, for instance).

[1] The local `./sbt` script simply guarantees that the sbt version is consistent with one that we know has all the proper libraries available.

[2] In `target/gen-scala/com/twitter/searchbird/SearchbirdService.scala`.

[3] See Ostrich's README for more information.

Built at @twitter by @stevej, @marius, and @lahosken with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman, and @mccv; Russian translation by appigram; Chinese simple translation by jasonqu; Korean translation by enshahar;