Replication, Clustering, and Administration

Scaling CouchDB

O'REILLY®

Bradley Holt

Scaling CouchDB

This practical guide offers a short course on scaling CouchDB to meet the capacity needs of your distributed application. Through a series of scenario-based examples, this book lets you explore several methods for creating a system that can accommodate growth and meet expected demand. In the process, you learn about several tools that can help you with replication, load balancing, clustering, and load testing and monitoring.

- Apply performance tips for tuning your database
- Replicate data, using Futon and CouchDB's RESTful interface
- Distribute CouchDB's workload through load balancing
- Learn options for creating a cluster of CouchDB nodes, including BigCouch, Lounge, and Pillow
- Conduct distributed load testing with Tsung

Strata
Making Data Work

Strata is the emerging ecosystem of people, tools, and technologies that turn big data into smart decisions. Find information and resources at oreilly.com/data.

Twitter: @oreillymedia
facebook.com/oreilly

US $29.99 CAN $34.99
ISBN: 978-1-449-30343-3

O'REILLY®
oreilly.com

Scaling CouchDB

Scaling CouchDB

Replication, Clustering, and Administration

Bradley Holt

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Scaling CouchDB
by Bradley Holt

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: Kristen Borg
Proofreader: Kristen Borg

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
April 2011: First Edition.

ISBN: 978-1-449-30343-3

[LSI]

1300886680

Table of Contents

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

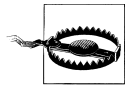
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does

require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Scaling CouchDB* by Bradley Holt (O'Reilly). Copyright 2011 Bradley Holt, 978-1-449-30343-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9781449303433

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I'd first like to thank Damien Katz, creator of CouchDB, and all of CouchDB's contributors. I'd also like to thank all of the contributors to the other open source software tools referenced in this book. This includes contributors to the Apache HTTP Server, BigCouch, CouchDB Lounge, Pillow, and Tsung. Knut Ola Hellan (creator of Pillow) and Martin Brown (from Couchbase) both provided valuable feedback which helped to make this book better. Mike Loukides, this book's editor, and the rest of the team at O'Reilly Media were very responsive and helpful.

CHAPTER 1

Defining Scaling Goals

Before you can scale CouchDB, you need to define your scaling goals. Once you have defined your goals then you can design your system. You should test the scalability of your system before it is deployed. See Chapter 6 for information about how to perform distributed load testing on your system using Tsung. When your system has been deployed to production, you should continue to monitor its performance and resource utilization using a tool such as Munin (a CouchDB plugin for Munin is available at *https://github.com/strattg/munin-plugin-couchdb*) or Nagios. You can monitor an individual node by issuing a `GET` HTTP request to `/_stats` which will return various statistics about the CouchDB node.

What is Scalability?

It's important to note the distinctions between performance, vertical scaling, and horizontal scaling. Performance typically refers to properties of a system such as response time or throughput. Vertical scaling (or scaling up) means adding computing capacity to a single node in a system. This could be through added memory, a faster CPU, or larger hard drives. While memory gets cheaper, CPUs get faster, and hard drives get larger every year, at any given moment there's an upward limit to vertical scaling. Also, the hardware for high capacity nodes is usually more expensive per unit of computing capacity (as defined by your application) than commodity hardware.

When someone uses the word "scalability," they are often referring to horizontal scalability. Horizontal scaling (or scaling out) is when a system's computing capacity is scaled by adding new nodes. In theory, each new node adds the entire amount of that node's computing capacity to the system. In practice, true horizontal scalability is rarely, if ever, achieved. The network overhead of nodes communicating with each other can detract from the overall computing capacity of the system. Also, few systems are configured such that there is no redundant computing work between nodes.

Capacity Planning

Capacity planning is often lumped together with scaling. With capacity planning, the focus is on creating a system that can meet an expected amount of demand. With scalability, the focus is on creating a system that is capable of accommodating growth. As mentioned before, both vertical and horizontal scaling each have their own limitations. Given these limitations, a combination of scalability and capacity planning is often warranted. Put another way, define a maximum capacity to which you need to scale and test so that your system can scale to meet that capacity—there is no such thing as infinite scalability, and attempting to create such a system would be prohibitively expensive. In Chapter 6, we will take a look at testing the capacity of your system through distributed load testing.

The CAP Theorem

CouchDB, like any database, balances three different concerns: *consistency*, *availability*, and *partition tolerance* (see Chapter 2 in CouchDB: The Definitive Guide [O'Reilly]). By design, CouchDB focuses on availability and partition tolerance and gives up consistency in exchange.

Consistency

Consistency is a database property whereby all clients will always see a consistent view of the data in your database. This will be true even during concurrent updates. Once you have more than one CouchDB node, you will typically give up consistency in exchange for *eventual consistency*. Through *replication*, all CouchDB nodes can eventually be made to have a consistent view of the data. How quickly do you need your data replicated? Do some nodes need to be consistent immediately?

Availability

Through *load balancing*, CouchDB can achieve a high level of availability. This means that a large number of requests can be served concurrently while still providing all clients with access to create, read, update, and delete data. How many requests per second does your system need to be capable of handling? What will be the ratio of read requests (`GET`) to write requests (`POST`, `PUT`, `DELETE`)?

Partition Tolerance

CouchDB takes a peer-to-peer approach to replication giving it the property of partition tolerance. Multiple CouchDB nodes can store copies of your data. CouchDB nodes can operate in a "split-brain" scenario where the nodes are disconnected from each other and thus cannot be replicated, but will be replicated once a connection is re-established. How often and for how long will your CouchDB nodes be disconnected from each other? For example, will you have CouchDB instances on mobile devices that are occasionally connected? Will you be hosting CouchDB nodes in separate data centers? Are you dealing with a Big Data problem where you need to store a dataset that is too big for a single node?

CHAPTER 2

Tuning and Designing for Scale

In this chapter, we will take a look at some performance tips that you can apply when tuning your database. While not directly related to scalability, increasing performance can increase the overall capacity of your system. There are many options available when tuning CouchDB to meet your needs.

We will also discuss considerations around the design of your documents. CouchDB is a schema-less database, giving you much flexibility in designing the document boundaries for your data. However, the decisions you make around designing your documents can have an impact on the performance and scalability of your database.

Performance Tips

The best way to increase the capacity of your database is to not send requests to it in the first place. Sometimes you can't forego a database request altogether, but you can limit the amount of work you ask the database to do. Here are a some tips to limit the amount of work you ask of CouchDB and to increase performance (the applicability of these tips to your application may vary):

For more information, see Chapter 23 in CouchDB: The Definitive Guide (O'Reilly), Operating CouchDB, and Operating CouchDB II.

- Cache documents and query results using memcached or another caching system. For systems under heavy load, even caches that expire after only a few seconds can save many extra requests to your database. This caching can be done in your application or by using a reverse proxy server that supports caching. Optionally, you can use the `/<db>/_changes` API (replacing `<db>` with the name of your database) to watch for deleted documents and/or new document revisions so that you can immediately invalidate the cache, instead of waiting for the cache to expire.

- Use `Etag`s to send conditional requests. The response to every `GET` request to a document or view includes an `Etag` HTTP header (for documents, the `Etag`'s value is the document's revision in double quotes). Cache the document or view results along with its corresponding `Etag`. Send an `If-None-Match` HTTP header containing the `Etag`'s value with subsequent requests to the same URL (`Etag`s are only valid for a given URL). If the document or view results have not changed, then CouchDB won't need to send the entire response to you and will instead respond with a `304 Not Modified` HTTP response code, a `Content-Length` of `0`, and no message-body.
- Don't use the `include_docs` parameter (or set it to `false`) when querying views. While convenient, including documents can cause performance issues. There are two alternatives. First, you can emit the entire document as the value in your Map function, for instance, `emit(key, doc)`. This will increase the size of your index, but will use less I/O resources and result in faster document retrieval. Second, you can make a separate request for each document. Assuming you have cached your documents, then some percentage of these requests will result in cache hits.
- Don't use random document IDs. CouchDB will perform best with document IDs that are mostly monotonic (in simpler terms, mostly sequential). This has to do with the B-tree (technically B+tree) structure that CouchDB uses to store data. The simplest way to generate mostly monotonic document IDs is to use the default value of `sequential` for the `algorithm` option in the `uuids` configuration section and let CouchDB generate your document IDs.
- Use the bulk documents API, if possible. CouchDB allows you to POST a collection of documents to `/<db>/_bulk_docs` (replacing `<db>` with the name of your database). Inserting documents in bulk can be many times faster than individual inserts. CouchDB supports a `non-atomic` (the default) or an `all-or-nothing` model for bulk updates. Using the `non-atomic` model, some documents may be updated and some may not. Documents may fail to update due to a document update conflict, or because of a power failure. Under the `all-or-nothing` model, either all of the documents will be updated, or none of them will be updated. Instead of causing a document update conflict, an update to a document using the non-latest revision will result in the document being written, but also being marked as conflicted. If there is a power failure before all of the documents can be written, then on restart none of the documents will have been saved.
- Use batch mode to speed up writes. In this mode, CouchDB will save documents in memory and flush them to disk in batches. This can be triggered by setting the `batch` query parameter to `ok` when doing a `POST` or `PUT` of a document. Since there is no guarantee that your document will be written to disk, CouchDB returns an HTTP response of `202 Accepted` instead of `201 Created`. Batch mode may not be a good fit for your application.
- Leave delayed commits on. That is, leave the `delayed_commits` option set to `true` in the `couchdb` configuration section. Delayed commits means that CouchDB will

not trigger an fsync after each write (an fsync commits buffered data to disk). For non-bulk writes, requiring an fsync will have a major performance impact.

- If you can live without the most up-to-date view results, set the `stale` query parameter's value to `ok` when querying views. If documents have been updated, this saves CouchDB from recomputing views on each read. You will need to have a system for querying these views without the `stale` query parameter or else your views will never get updated. This can be done with a cron job or through another automated process. You can use the `HEAD` HTTP method for these requests to save bandwidth.
- Run database compaction, view compaction, and view cleanup when the database is not under heavy load. Schedule these processes to happen during lower usage periods.
- Spread your views across multiple design documents. This should speed up view compaction and cleanup operations.
- Instead of paginating view results using the `skip` parameter, use the `startkey` and `startkey_docid` parameters (`endkey` and `endkey_docid` if output is reversed). When skipping a large number of rows, CouchDB still needs to scan the entire B-tree index, starting from the `startkey` and `startkey_docid` (if specified). Set the `skip` parameter's value to `1` and use the key of the last row on the previous page as the `startkey` parameter (`endkey` if output is reversed), and the document ID of the last row on the previous page as the `startkey_docid` parameter (`endkey_docid` if output is reversed).
- While it may seem obvious, it's worth mentioning that faster disks, more CPUs, and more memory should increase the performance of CouchDB. Being a database, CouchDB is ultimately limited to your disk I/O throughput. Higher RPM drives, solid-state drives (SSD), or a RAID configuration (the appropriate RAID level depends on your needs related to data redundancy, read performance, and write performance) could help. CouchDB (due to its Erlang underpinnings) can take advantage of multiple CPUs. The performance of view queries can be improved by having more RAM available with which CouchDB can cache views' B-tree indices.

Systems can behave very differently under load than they do otherwise. What performs well with only one node may perform very differently when scaled out to multiple nodes. Performance tuning of large scale systems can sometimes result in counterintuitive optimizations.

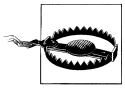
If you are experiencing performance or scaling problems, be sure that your database is, in fact, the main source of your problems. Likely you have an application that is the main (if not the only) client to your database. If that application is a web application, then it has clients connecting to it through web browsers. Each tier of your application is a potential bottleneck. You will want to make sure that you are focusing your efforts on the right part of your system.

Document Design

The document boundaries of your data can have a significant impact on the ability of your system to scale. Design your documents around transaction, distribution, and concurrency boundaries. Let's use two extreme scenarios to illustrate the point.

Transaction, distribution, and concurrency boundaries are also used in domain-driven design when defining *Aggregates*. For more information, see Eric Evans' book *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley), Chapter Six: The Life Cycle of a Domain Object.

In the first contrived scenario, you could put all of your data in one document (let's ignore, for a moment, how large and awkward this document would become). Within a single CouchDB node, an update of a single document is transactional. Putting all of your data in one document would make all operations transactional. However, clients would often get document update conflicts. Replication would almost always result in conflicts. Following is an example of putting all data in one document:



The following document is an intentionally bad design. Do not structure your CouchDB database this way!

```
{
   "_id":"catalog",
   "978-0-596-80579-1":{
      "title":"Building iPhone Apps with HTML, CSS, and JavaScript",
      "subtitle":"Making App Store Apps Without Objective-C or Cocoa",
      "authors":[
         "Jonathan Stark"
      ],
      "publisher":"O'Reilly Media",
      "formats":[
         "Print",
         "Ebook",
         "Safari Books Online"
      ],
      "released":"2010-01-08",
      "pages":192
   },
   "978-0-596-15589-6":{
      "title":"CouchDB: The Definitive Guide",
      "subtitle":"Time to Relax",
      "authors":[
         "J. Chris Anderson",
         "Jan Lehnardt",
         "Noah Slater"
      ],
```

```
      "publisher":"O'Reilly Media",
      "released":"2010-01-19",
      "pages":272,
      "formats":[
         "Print",
         "Ebook",
         "Safari Books Online"
      ]
   },
   "978-1-565-92580-9":{
      "title":"DocBook: The Definitive Guide",
      "authors":[
         "Norman Walsh",
         "Leonard Muellner"
      ],
      "publisher":"O'Reilly Media",
      "formats":[
         "Print"
      ],
      "released":"1999-10-28",
      "pages":648
   },
   "978-0-596-52926-0":{
      "title":"RESTful Web Services",
      "subtitle":"Web services for the real world",
      "authors":[
         "Leonard Richardson",
         "Sam Ruby"
      ],
      "publisher":"O'Reilly Media",
      "released":"2007-05-08",
      "pages":448,
      "formats":[
         "Print",
         "Ebook",
         "Safari Books Online"
      ]
   }
}
```

In the second contrived scenario, you could put each discrete piece of data in its own document, much like a normalized relational database. Clients would get document update conflicts less often and replication would generate fewer conflicts. However, related data could often be disjointed since CouchDB does not support transactions across document boundaries. Following is an example of breaking one document into smaller pieces of data. First, a "catalog" document:

```
{
   "_id":"catalog/978-0-596-80579-1",
   "collection":"catalog",
   "title":"Building iPhone Apps with HTML, CSS, and JavaScript",
   "subtitle":"Making App Store Apps Without Objective-C or Cocoa",
   "authors":[
      "author/3840"
   ],
```

```
    "publisher":"publisher/oreilly",
    "formats":[
       "format/book",
       "format/ebook",
       "format/saf"
    ],
    "released":"2010-01-08",
    "pages":192
}
```

The "author" document for `author/3840`:

```
{
   "_id":"author/3840",
   "collection":"author",
   "name":"Jonathan Stark",
}
```

The "publisher" document for `publisher/oreilly`:

```
{
   "_id":"publisher/oreilly",
   "collection":"publisher",
   "name":"O'Reilly Media",
}
```

The "format" document for `format/book`:

```
{
   "_id":"format/book",
   "collection":"format",
   "name":"Print",
}
```

The "format" document for `format/ebook`:

```
{
   "_id":"format/ebook",
   "collection":"format",
   "name":"Ebook",
}
```

The "format" document for `format/saf`:

```
{
   "_id":"format/saf",
   "collection":"format",
   "name":"Safari Books Online",
}
```

The first approach gives you a high level of consistency but reduces availability (clients will get document update conflicts more often) and reduces partition tolerance (replication will often lead to conflicts). The second approach may reduce consistency but can give you a higher level of availability and partition tolerance.

CHAPTER 3
Replication

Replication in CouchDB is peer-based and bi-directional, although any given replication process is one-way, from the *source* to the *target*. Replication can be run from Futon, CouchDB's web administration console, or by sending a `POST` request to `_replicate` containing a JSON object with replication parameters. Let's assume we have two databases, both running on the same CouchDB node, that we want to replicate: `catalog-a` and `catalog-b` (we can also replicate databases on different CouchDB nodes).

Using Futon:

1. Navigate to *http://localhost:5984/_utils/* using your web browser.
2. Create the `catalog-a` and `catalog-b` databases.
3. Create a new, empty document (with only an `_id` field) in the `catalog-a` database.
4. Under "Tools," click "Replicator."
5. Under "Replicate changes from," leave "Local database" selected, and select "catalog-a."
6. Under "to," leave "Local database" selected, and select "catalog-b."
7. Click the "Replicate" button. Figure 3-1 shows how everything should look (the details under "Event" will be different for you). Optionally, you could have checked "Continuous" to trigger continuous replication.

If you would prefer to use cURL, first create the `catalog-a` database:

```
curl -X PUT http://localhost:5984/catalog-a
```

The response:

```
{"ok":true}
```

Create the `catalog-b` database:

```
curl -X PUT http://localhost:5984/catalog-b
```

The response:

```
{"ok":true}
```

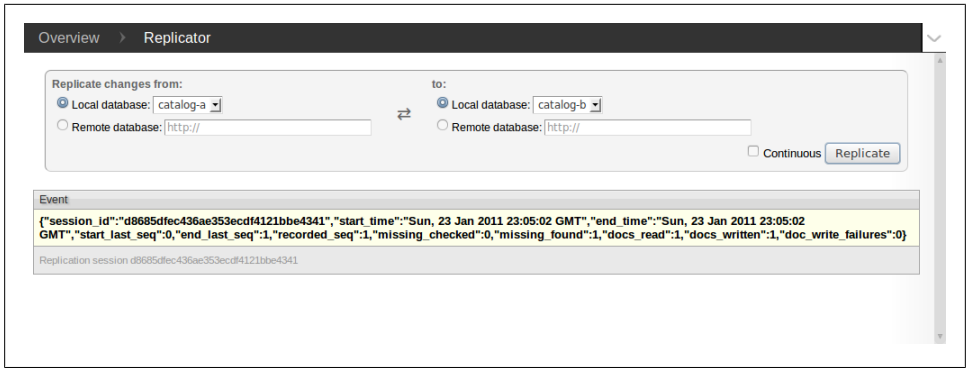


Figure 3-1. Replicating from catalog-a to catalog-b using Futon

Create a new, empty, document in the `catalog-a` database:

```
curl -X POST http://localhost:5984/catalog-a \
-H "Content-Type: application/json" \
-d '{}'
```

The response (your `id` and `rev` values will be different):

```
{
   "ok":true,
   "id":"a6b0e79a4a9be7359e9e83c52100 2cac",
   "rev":"1-967a00dff5e02add41819138abb3284d"
}
```

Replicate from `catalog-a` to `catalog-b`:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-a",
   "target":"catalog-b"
}'
```

The response (your details will be different):

```
{
   "ok":true,
   "session_id":"baf54178bc255b76a4a572ccbc67edcc",
   "source_last_seq":1,
   "history":[
      {
         "session_id":"baf54178bc255b76a4a572ccbc67edcc",
         "start_time":"Sun, 23 Jan 2011 23:48:18 GMT",
         "end_time":"Sun, 23 Jan 2011 23:48:18 GMT",
         "start_last_seq":0,
         "end_last_seq":1,
         "recorded_seq":1,
         "missing_checked":0,
         "missing_found":1,
```

```
            "docs_read":1,
            "docs_written":1,
            "doc_write_failures":0
         }
      ]
   }
```

To cancel replication:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-a",
   "target":"catalog-b",
   "cancel":true
}'
```

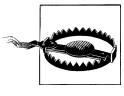
If you have a large number of documents, then you could potentially have a time consuming replication process. This is one use case for canceling a replication. Another use case is to cancel a continuous replication process.

The response (your details will be different):

```
{
   "ok":true,
   "_local_id":"ca1f51594051677b5a5af9bfa1ec0b01"
}
```

To replicate from `catalog-a` to `catalog-b` continuously:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-a",
   "target":"catalog-b",
   "continuous":true
}'
```



CouchDB 1.0.2 does not persist continuous replication between restarts. If CouchDB is restarted, then you must start continuous replication again. Permanent continuous replication is planned for a future version of CouchDB. For this reason, you may want to consider triggering continuous replication from within a cron job. Attempting to start continuous replication while continuous replication is already running is harmless.

The response (your details will be different):

```
{
   "ok":true,
   "_local_id":"ca1f51594051677b5a5af9bfa1ec0b01"
}
```

To cancel continuous replication:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-a",
   "target":"catalog-b",
   "continuous":true,
   "cancel":true
}'
```

The response (your details will be different):

```
{
   "ok":true,
   "_local_id":"ca1f51594051677b5a5af9bfa1ec0b01"
}
```

Filters and Specifying Documents

Sometimes you do not want to replicate every document in your database. Filter functions can be used to determine which documents should and should not be replicated. A filter function will simply return `true` if the given document should be replicated, and `false` if the given document should *not* be replicated. A filter function can be defined within a design document on the source database. When initiating replication, you can specify which filter function, from which design document, to use.

Here is an example of a filter function that would cause only documents with a `collection` value of `author` to be replicated:

```
function(doc, req) { return "author" == doc.collection }
```

The first parameter, `doc` in the above example, is a candidate document for replication. The second parameter, `req` in the above example, is the replication request. This second parameter contains the details of the replication request including the HTTP headers, HTTP method, and query parameters. See the Filters section in "What's new in Apache CouchDB 0.11 — Part Three: New Features in Replication" at *http://blog.couchone.com/post/468392274/whats-new-in-apache-couchdb-0-11-part-three-new*. For example, `req.query` is a JSON object of query parameters (`query_params` from the replication request).

Each design document can have multiple named filters. Here is a design document containing the above filter, given the name `authors`:

```
{
   "_id":"_design/default",
   "filters":{
      "authors":"function(doc, req) { return \"author\" == doc.collection }"
   }
}
```

To use this filter during replication (the `create_target` parameter tells CouchDB to create the target database before replication):

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"oreilly",
   "target":"authors",
   "create_target":true,
   "filter":"default/authors"
}'
```

The response:

```
{
   "ok":true,
   "session_id":"2f34911e5e21dc3736Od1f33cc85fffa",
   "source_last_seq":3,
   "history":[
      {
         "session_id":"2f34911e5e21dc3736Od1f33cc85fffa",
         "start_time":"Sun, 30 Jan 2011 21:49:04 GMT",
         "end_time":"Sun, 30 Jan 2011 21:49:06 GMT",
         "start_last_seq":0,
         "end_last_seq":3,
         "recorded_seq":3,
         "missing_checked":0,
         "missing_found":1,
         "docs_read":1,
         "docs_written":1,
         "doc_write_failures":0
      }
   ]
}
```

We could parameterize our filter function as follows:

```
function(doc, req) { return req.query.collection == doc.collection }
```

Here's what the updated filter function would look like in a design document:

```
{
   "_id":"_design/default",
   "filters":{
      "collection":"function(doc, req) { return req.query.collection == doc.collection }"
   }
}
```

To use this new, parameterized, filter:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"oreilly",
   "target":"publishers",
   "create_target":true,
   "filter":"default/collection",
   "query_params":{
      "collection":"publisher"
   }
}'
```

The response:

```
{
   "ok":true,
   "session_id":"585b3ecae989739796b5d161c28535fc",
   "source_last_seq":4,
   "history":[
      {
         "session_id":"585b3ecae989739796b5d161c28535fc",
         "start_time":"Sun, 30 Jan 2011 21:55:28 GMT",
         "end_time":"Sun, 30 Jan 2011 21:55:28 GMT",
         "start_last_seq":0,
         "end_last_seq":4,
         "recorded_seq":4,
         "missing_checked":0,
         "missing_found":1,
         "docs_read":1,
         "docs_written":1,
         "doc_write_failures":0
      }
   ]
}
```

If you know the exact IDs of the documents you want to replicate, then you can specify those instead of using a filter:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"oreilly",
   "target":"formats",
   "create_target":true,
   "doc_ids":[
      "format/saf",
      "format/ebook",
      "format/book"
   ]
}'
```

The response:

```
{
   "ok":true,
   "start_time":"Sun, 30 Jan 2011 22:05:28 GMT",
   "end_time":"Sun, 30 Jan 2011 22:05:28 GMT",
   "docs_read":3,
   "docs_written":3,
   "doc_write_failures":0
}
```

Conflict Resolution

When replicating, you will inevitably run into document update conflicts. This can happen when a document with the same ID has been updated independently on two or more CouchDB nodes. To handle document update conflicts, you will need to create a view to find conflicts. Here is the Map function that will let us find conflicts:

```
function(doc) {
   if (doc._conflicts) {
       for (var i in doc._conflicts) {
           emit(doc._conflicts[i]);
       }
   }
}
```

Let's save this to a design document:

```
curl -X PUT http://localhost:5984/catalog-a/_design/default -d \
'{
   "_id": "_design/default",
   "language": "javascript",
   "views": {
       "conflicts": {
           "map":
"function(doc) {
   if (doc._conflicts) {
       for (var i in doc._conflicts) {
           emit(doc._conflicts[i]);
       }
   }
}"       }
   }
}'
```

The response:

```
{
   "ok":true,
   "id":"_design/default",
   "rev":"1-05df67108309b2783233f27ffff31c59"
}
```

Now let's create a conflict. First, create the following document within the `catalog-a` database:

```
curl -X PUT http://localhost:5984/catalog-a/978-0-596-80579-1 \
-H "Content-Type: application/json" \
-d '{
   "title":"Building iPhone Apps with HTML, CSS, and JavaScript",
}'
```

The response:

```
{
   "ok":true,
   "id":"978-0-596-80579-1",
   "rev":"1-8e68b2b2f14a81190889dab9d04481d2"
}
```

Next, create the same document within the `catalog-b` database, but with a slightly different title (the comma is missing after "CSS"):

```
curl -X PUT http://localhost:5984/catalog-b/978-0-596-80579-1 \
-H "Content-Type: application/json" \
-d '{
   "title":"Building iPhone Apps with HTML, CSS and JavaScript",
}'
```

The response:

```
{
   "ok":true,
   "id":"978-0-596-80579-1",
   "rev":"1-25042aaa901375bfd7cb63d189275197"
}
```

Replicate from `catalog-a` to `catalog-b`:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-a",
   "target":"catalog-b"
}'
```

Replicate back from `catalog-b` to `catalog-a` (so that the conflict will exist in both databases):

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-b",
   "target":"catalog-a"
}'
```

Querying the conflicts view from `catalog-a` or from catalog-b should return the same results:

```
curl -X GET http://localhost:5984/catalog-a/_design/default/_view/conflicts

curl -X GET http://localhost:5984/catalog-b/_design/default/_view/conflicts
```

The response to both queries:

```
{
   "total_rows":1,
   "offset":0,
   "rows":[
      {
         "id":"978-0-596-80579-1",
         "key":"1-25042aaa901375bfd7cb63d189275197",
         "value":null
      }
   ]
}
```

CouchDB uses a deterministic algorithm to pick the winner when there is a conflict. Every CouchDB node will always pick the same winner of any given conflict. The algorithm it uses is quite simple. First, CouchDB looks at the incrementing part of the revision number (the part before the "-") and the document with the highest number wins. If both documents have the same number of revisions, then CouchDB simply does an ASCII comparison of the revision number and the document with the highest sort order wins. It is always a good idea to handle conflicts within your application by automatically merging documents in a way that makes sense to your application, or by presenting the conflict to an end user to resolve.

This tells us that the document with an ID of `978-0-596-80579-1` has a conflict, and that revision `1-25042aaa901375bfd7cb63d189275197` lost in the conflict resolution. If you'd like, you can verify this by querying the conflicted document on `catalog-a`, setting the `conflicts` query parameter to `true`:

```
curl -X GET http://localhost:5984/catalog-a/978-0-596-80579-1?conflicts=true
```

The response:

```
{
   "_id":"978-0-596-80579-1",
   "_rev":"1-8e68b2b2f14a81190889dab9d04481d2",
   "title":"Building iPhone Apps with HTML, CSS, and JavaScript",
   "_conflicts":[
      "1-25042aaa901375bfd7cb63d189275197"
   ]
}
```

Based on this response, we can tell that revision `1-8e68b2b2f14a81190889dab9d04481d2` (the one *with* the comma) was picked by CouchDB as the winner. If we take a look at the `_conflicts` array, we can see that it won out against revision `1-25042aaa901375bfd7cb63d189275197` (the one *without* the comma). Let's examine revision `1-25042aaa901375bfd7cb63d189275197`:

```
curl -X GET http://localhost:5984/catalog-a/978-0-596-80579-1\
?rev=1-25042aaa901375bfd7cb63d189275197
```

The response:

```
{
   "_id":"978-0-596-80579-1",
   "_rev":"1-25042aaa901375bfd7cb63d189275197",
   "title":"Building iPhone Apps with HTML, CSS and JavaScript"
}
```

We can see here that revision `1-25042aaa901375bfd7cb63d189275197` is, in fact, the one with the missing comma. We now have three choices:

1. Pick revision `1-8e68b2b2f14a81190889dab9d04481d2` as the winner (the one CouchDB already picked).
2. Pick revision `1-25042aaa901375bfd7cb63d189275197` as the winner.
3. Merge both revisions into a new revision.

Let's take a look at each option. Since each is mutually exclusive, you will need to pick one if you are following along (or start this example again from the beginning for each).

Picking the Same Revision as CouchDB

If you pick the same revision to win as CouchDB picked, then you could just do nothing. However, the conflict will continue to be listed and, assuming you have an automated process to deal with conflicts, your application will continue to deal with the conflict indefinitely. To clear the conflict, you can simply delete the revision you don't like:

```
curl -X DELETE http://localhost:5984/catalog-a/978-0-596-80579-1 \
-H "If-Match: 1-25042aaa901375bfd7cb63d189275197"
```

The response:

```
{
   "ok":true,
   "id":"978-0-596-80579-1",
   "rev":"2-638024b9b98a9d951519b0bf7e95953e"
}
```

Whenever CouchDB deletes a document, it creates a new revision with `_deleted` field set to `true`. The same thing happens when you delete a conflicted revision.

Let's take a look at our conflicts view to make sure that the conflict has, in fact, been resolved:

```
curl -X GET http://localhost:5984/catalog-a/_design/default/_view/conflicts
```

The response shows us that there are now no conflicts:

```
{
   "total_rows":0,
   "offset":0,
   "rows":[

   ]
}
```

Let's verify that the document has not been deleted:

```
curl -X GET http://localhost:5984/catalog-a/978-0-596-80579-1
```

The response:

```
{
   "_id":"978-0-596-80579-1",
   "_rev":"1-8e68b2b2f14a81190889dab9d04481d2",
   "title":"Building iPhone Apps with HTML, CSS, and JavaScript"
}
```

Be sure to replicate your changes to the catalog-b database:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-a",
   "target":"catalog-b"
}'
```

Let's double-check that the conflict has also been resolved in the `catalog-b` database:

```
curl -X GET http://localhost:5984/catalog-b/_design/default/_view/conflicts
```

The response:

```
{
   "total_rows":0,
   "offset":0,
   "rows":[

   ]
}
```

Picking a Conflicted Revision

In this scenario, we don't agree with CouchDB's pick and want to instead pick a conflicted revision to win. Like in the previous scenario, we simply delete the revision we don't like (which happens to be the revision that CouchDB had picked as the winner):

```
curl -X DELETE http://localhost:5984/catalog-a/978-0-596-80579-1 \
-H "If-Match: 1-8e68b2b2f14a81190889dab9d04481d2"
```

The response:

```
{
   "ok":true,
   "id":"978-0-596-80579-1",
   "rev":"2-3baff9c19dbcb58cf81fdc9a21602a0c"
}
```

Let's take a look at our conflicts view to make sure that the conflict has, in fact, been resolved:

```
curl -X GET http://localhost:5984/catalog-a/_design/default/_view/conflicts
```

The response shows us that there are now no conflicts:

```
{
   "total_rows":0,
   "offset":0,
   "rows":[

   ]
}
```

Let's verify that the document has not been deleted:

```
curl -X GET http://localhost:5984/catalog-a/978-0-596-80579-1
```

The response:

```
{
   "_id":"978-0-596-80579-1",
   "_rev":"1-25042aaa901375bfd7cb63d189275197",
   "title":"Building iPhone Apps with HTML, CSS and JavaScript"
}
```

Be sure to replicate your changes to the catalog-b database:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-a",
   "target":"catalog-b"
}'
```

Let's double-check that the conflict has also been resolved in the `catalog-b` database:

```
curl -X GET http://localhost:5984/catalog-b/_design/default/_view/conflicts
```

The response:

```
{
   "total_rows":0,
   "offset":0,
   "rows":[

   ]
}
```

Merging Revisions

In this scenario, we will create a new revision, merging properties of both documents, and then delete the conflicting document. First, let's update the document, merging our changes (our "merge" involves leaving out the comma but using an ampersand instead of "and"):

```
curl -X PUT http://localhost:5984/catalog-a/978-0-596-80579-1 \
-H "Content-Type: application/json" \
-H "If-Match: 1-8e68b2b2f14a81190889dab9d04481d2" \
-d '{
   "title":"Building iPhone Apps with HTML, CSS & JavaScript",
}'
```

The response:

```
{
   "ok":true,
   "id":"978-0-596-80579-1",
   "rev":"2-f515785a36225fd7511ad756aa1d3bc0"
}
```

Next, we need to delete the conflicted version:

```
curl -X DELETE http://localhost:5984/catalog-a/978-0-596-80579-1 \
-H "If-Match: 1-25042aaa901375bfd7cb63d189275197"
```

The response:

```
{
   "ok":true,
   "id":"978-0-596-80579-1",
   "rev":"2-638024b9b98a9d951519b0bf7e95953e"
}
```

Let's take a look at our conflicts view to make sure that the conflict has been resolved:

```
curl -X GET http://localhost:5984/catalog-a/_design/default/_view/conflicts
```

The response shows us that there are now no conflicts:

```
{
   "total_rows":0,
   "offset":0,
   "rows":[

   ]
}
```

Let's verify that the document has not been deleted:

```
curl -X GET http://localhost:5984/catalog-a/978-0-596-80579-1
```

The response:

```
{
   "_id":"978-0-596-80579-1",
   "_rev":"2-f515785a36225fd7511ad756aa1d3bc0",
   "title":"Building iPhone Apps with HTML, CSS & JavaScript"
}
```

Be sure to replicate your changes to the catalog-b database:

```
curl -X POST http://localhost:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"catalog-a",
   "target":"catalog-b"
}'
```

Let's double-check that the conflict has also been resolved in the `catalog-b` database:

```
curl -X GET http://localhost:5984/catalog-b/_design/default/_view/conflicts
```

The response:

```
{
   "total_rows":0,
   "offset":0,
   "rows":[

   ]
}
```

CHAPTER 4
Load Balancing

Load balancing allows you to distribute the workload evenly across multiple CouchDB nodes. Since CouchDB uses an HTTP API, standard HTTP load balancing software or hardware can be used. With simple load balancing, each CouchDB node will maintain a full copy of your database through replication. Each document will eventually need to be written to every node, which is a limitation of this approach since the sustained write throughput of your entire system will be limited to that of the slowest node. You could replicate only certain documents using filter functions or by specifying document IDs, as discussed in Chapter 3. This approach to clustering could get complicated very quickly. See Chapter 5 for details on an alternative way to distribute your data across multiple CouchDB nodes.

In this scenario, we will set up a write-only master node and three read-only slave nodes. We will send all "unsafe" HTTP write requests (`POST`, `PUT`, `DELETE`, `MOVE`, and `COPY`) to the master node and load balance all "safe" HTTP read requests (`GET`, `HEAD`, and `OPTIONS`) across the three slave nodes. We will set up continuous replication from the write-only master to each of the read-only slave nodes. See Figure 4-1 for a diagram of the configuration we will be creating in this chapter.

`MOVE` and `COPY` are non-standard HTTP methods. Only versions 0.8 and 0.9 of CouchDB supported the `MOVE` HTTP method. The `MOVE` HTTP method was removed from CouchDB since it was really just a `COPY` followed by a `DELETE`, but implied that there was a transaction across these two operations (which there was not). Assuming you are using a newer version of CouchDB, then there's no need to concern yourself with the `MOVE` HTTP method.

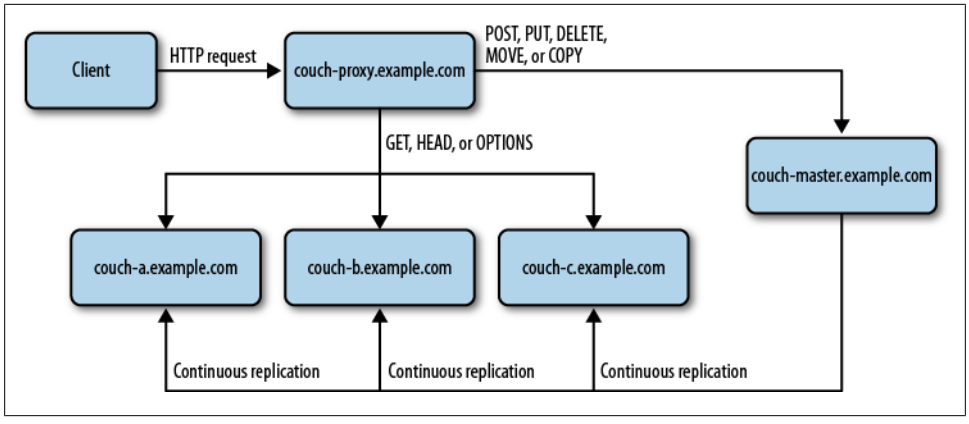
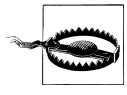


Figure 4-1. Load balancing configuration

There are many load balancing software and hardware options available. A full discussion of all the available tools and how to install and configure each on every available platform is beyond the scope of this book. Instead, we will focus on installing and configuring the Apache HTTP Server as a load balancer. We'll be using Ubuntu, but these instructions should be easily adaptable to your operating system.



You may want to consider having multiple load balancers so that you can remove the load balancer as a single point of failure. This setup typically involves having two or more load balancers sharing the same IP address, with one configured as a failover. The details of this configuration are beyond the scope of this book.

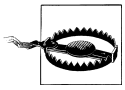
Apache was chosen as the load balancer for this scenario because it is relatively easy to configure and has the basic capabilities needed. Other load balancers you may want to consider are HAProxy, Varnish, Pound, Perlbal, Squid, nginx, and Linux-HA (High-Availability Linux) on Linux Standard Base (LSB). This example illustrate a basic load balancing setup. Hardware load balancers are also available. Your hosting provider may also offer its own, proprietary load balancing tools. For example, Amazon has a tool called Elastic Load Balancing and Rackspace provides a service called Rackspace Cloud Load Balancers (in beta as of this writing).

CouchDB Nodes

In the following scenario, we will send write requests to one master node with a domain name of `couch-master.example.com` and distribute read requests to three nodes on machines with domain names of `couch-a.example.com`, `couch-b.example.com`, and `couch-c.example.com`. Install CouchDB on the master node and on all three slave nodes:

```
sudo aptitude install couchdb
```

We need to configure CouchDB to allow connections from the outside world. On all four nodes, configure CouchDB to bind to each server's IP address by editing the `[httpd]` section of */etc/couchdb/local.ini* as follows, replacing `<server IP address>` with your server's IP address:



Unless your server is behind a firewall, this configuration change will allow anyone to access your CouchDB database. You will likely want to configure authentication and authorization. For information on this, see Chapter 22 in CouchDB: The Definitive Guide (O'Reilly).

```
[httpd]
; port = 5984
bind_address = <server IP address>
; Uncomment next line to trigger basic-auth popup on unauthorized requests.
;WWW-Authenticate = Basic realm="administrator"
```

If your server has multiple IP addresses, you can use `0.0.0.0` as the `bind_address` to bind on all IP addresses.

Restart CouchDB on all four nodes:

```
sudo /etc/init.d/couchdb restart
```

Test all four nodes by trying to connect to each *from a different machine*:

```
curl -X GET http://couch-master.example.com:5984/
curl -X GET http://couch-a.example.com:5984/
curl -X GET http://couch-b.example.com:5984/
curl -X GET http://couch-c.example.com:5984/
```

The response to each request should be:

```
{"couchdb":"Welcome","version":"1.0.1"}
```

If you can't connect remotely to one or more of the CouchDB nodes, then double-check that the `bind_address` in */etc/couchdb/local.ini* is set to each machine's correct IP address, respectively, and that you have restarted CouchDB.

Create a database named `api` on all four nodes:

```
curl -X PUT http://couch-master.example.com:5984/api
curl -X PUT http://couch-a.example.com:5984/api
curl -X PUT http://couch-b.example.com:5984/api
curl -X PUT http://couch-c.example.com:5984/api
```

The response to each request should be:

```
{"ok":true}
```

At this point we have four CouchDB databases, on four different nodes, running independently. If we write data to the master node, it will not be replicated to any of the slave nodes yet.

Replication Setup

CouchDB supports both *pull replication* and *push replication*. Pull replication is when replication is triggered from the same node as the target. Push replication is when replication is triggered from the same node as the source.

Set up a pull replication with `couch-a.example.com` pulling changes from `couch-master.example.com`:

```
curl -X POST http://couch-a.example.com:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"http://couch-master.example.com:5984/api",
   "target":"api",
   "continuous":true
}'
```

The response (your details will be different):

```
{"ok":true,"_local_id":"471f59393820994cd50fb432b17c9a96"}
```

Set up a pull replication with `couch-b.example.com` pulling changes from `couch-master.example.com`:

```
curl -X POST http://couch-b.example.com:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"http://couch-master.example.com:5984/api",
   "target":"api",
   "continuous":true
}'
```

The response (your details will be different):

```
{"ok":true,"_local_id":"0fdc3a4b85e224e51fdb6ce63f9bcbc6"}
```

Set up a pull replication with `couch-c.example.com` pulling changes from `couch-master.example.com`:

```
curl -X POST http://couch-c.example.com:5984/_replicate \
-H "Content-Type: application/json" \
-d \
'{
   "source":"http://couch-master.example.com:5984/api",
   "target":"api",
   "continuous":true
}'
```

The response (your details will be different):

```
{"ok":true,"_local_id":"ed8941ac5abf1cd7370b2d9a79000a11"}
```

You may want to set up replication using a separate, private, network so that you can have dedicated bandwidth for private and public requests.

Proxy Server Configuration

The Apache HTTP server is extremely versatile. It has many features including a proxy server and a load balancer (as of version 2.1). See Recipe 10.9 in Apache Cookbook, Second Edition (O'Reilly). In this exercise, we will create our load balancer on a machine with a domain name of `couch-proxy.example.com`..

On `couch-proxy.example.com`, install Apache 2:

```
sudo aptitude install apache2
```

On `couch-proxy.example.com`, install `mod_proxy`:

```
sudo aptitude install libapache2-mod-proxy-html
```

On `couch-proxy.example.com`, enable `mod_proxy`:

```
sudo a2enmod proxy
```

On `couch-proxy.example.com`, enable `mod_proxy_http`:

```
sudo a2enmod proxy_http
```

On `couch-proxy.example.com`, enable `mod_proxy_balancer`:

```
sudo a2enmod proxy_balancer
```

We will also need `mod_headers` enabled:

```
sudo a2enmod headers
```

Finally, we will need `mod_rewrite` enabled:

```
sudo a2enmod rewrite
```

On `couch-proxy.example.com`, edit */etc/apache2/httpd.conf* and add the following (it is likely that the file will be empty to start with):

```
Header append Vary Accept
Header add Set-Cookie "NODE=%{BALANCER_WORKER_ROUTE}e; path=/api" \
env=BALANCER_ROUTE_CHANGED

<Proxy balancer://couch-slave>
    BalancerMember http://couch-a.example.com:5984/api route=couch-a max=4
    BalancerMember http://couch-b.example.com:5984/api route=couch-b max=4
    BalancerMember http://couch-c.example.com:5984/api route=couch-c max=4
    ProxySet stickysession=NODE
    ProxySet timeout=5
</Proxy>
```

```
RewriteEngine On
RewriteCond %{REQUEST_METHOD} ^(POST|PUT|DELETE|MOVE|COPY)$
RewriteRule ^/api(.*)$ http://couch-master.example.com:5984/api$1 [P]
RewriteCond %{REQUEST_METHOD} ^(GET|HEAD|OPTIONS)$
RewriteRule ^/api(.*)$ balancer://couch-slave$1 [P]
ProxyPassReverse /api http://couch-master:5984/api
ProxyPassReverse /api balancer://couch-slave
```

Apache allows for three possible load balancer scheduler algorithms. Traffic can be balanced based on number of requests (`lbmethod=byrequests`), the number of bytes transferred (`lbmethod=bytraffic`), or by the number of currently pending requests (`lbmethod=bybusyness`). The default is to balance by requests. To instead balance by busyness, add a `ProxySet lbmethod=bybusyness` directive to the end of the `<Proxy>` directive group (after `ProxySet timeout=5` and before `</Proxy>`), although the order doesn't matter.

You will also need to configure your virtual host to enable the rewrite engine and inherit the rewrite options from the server configuration above. Edit */etc/apache2/sites-enabled/000-default* (or the configuration file for the appropriate virtual host) and add the following before the closing `</VirtualHost>` directive group:

```
RewriteEngine On
RewriteOptions inherit
```

Let's take a look at each line of the */etc/apache2/httpd.conf* configuration file. The `Header append Vary Accept` line appends the value `Accept` to the `Vary` HTTP header. If you have `mod_deflate` enabled then this module will add a `Vary` HTTP header with a value of `Accept-Encoding`. A `Vary` HTTP header informs a client as to what set of request-header fields it is permitted to base its caching on. Since `mod_deflate` may be adding this header, and CouchDB uses the Accept header to vary the media type (reflected in a `Content-Type` header with either a value of `text/plain` or `application/json`), it's a good idea to make sure that clients know to also vary their caching based on the `Accept` HTTP header, and not just the `Accept-Encoding` HTTP header.

The line beginning with `Header add Set-Cookie` sets a cookie named `NODE` on the client. The value of this cookie will be the route name associated with the load balancer member that served the request. This allows for sticky sessions meaning that, once a client has been routed to a specific load balancer member, that client's requests will continue to be routed to that same load balancer member node. This provides more consistency to the client. The `path=/api` part indicates to the client the URL path for which the cookie is valid. The `env=BALANCER_ROUTE_CHANGED` part indicates that the cookie should only be sent if the load balancer route has changed.

The `<Proxy balancer://couch-slave>` directive group defines a load balancer named `couch-slave`. A later configuration directive will define what requests should be sent to this load balancer. The three `BalancerMember` lines each add a member to the load balancer. The `route=` parameters (e.g., `route=couch-a`) give each route a name. The route

name is used as the value of the NODE cookie. The max=4 parameters indicate the maximum number of connections that the proxy will allow to the backend server. The ProxySet stickysession=NODE directive indicates to the load balancer the cookie name to use (in this case NODE) when determining which route to use. The ProxySet timeout=5 directive instructs the proxy server to wait 5 seconds before timing out connections to the backend server.

Keep the maximum number of connections to each CouchDB node low (e.g., max=4). This will prevent each node from getting overloaded. While 4 may seem like a very low number, CouchDB will respond to each request very quickly and allow for a high level of throughput. If the proxy server has enough memory and is configured to allow enough concurrent clients itself, then it can effectively queue requests for the backend servers.

If we didn't need to proxy requests based on the HTTP method, we could have used the ProxyPass directive. However, for this added flexible we need to use mod_rewrite with the proxy ([P]) flag. The RewriteEngine On line enables the rewrite engine. The next line sets up a rewrite condition that says to only run the subsequent rewrite rule if the request HTTP method is POST, PUT, DELETE, MOVE, or COPY:

```
RewriteCond %{REQUEST_METHOD} ^(POST|PUT|DELETE|MOVE|COPY)$
```

The subsequent rewrite rule then proxies all requests to URIs starting with /api to the equivalent URI on http://couch-master.example.com:5984 (again, only if the previous rewrite condition has been met):

```
RewriteRule ^/api(.*)$ http://couch-master.example.com:5984/api$1 [P]
```

The next line contains another rewrite condition. This one says to only run the subsequent rewrite rule if the request HTTP method is GET, HEAD, or OPTIONS:

```
RewriteCond %{REQUEST_METHOD} ^(GET|HEAD|OPTIONS)$
```

The subsequent rewrite rule then proxies all requests to URIs starting with /api to the equivalent URI on the couch-master load balancer (again, only if the previous rewrite condition has been met):

```
RewriteRule ^/api(.*)$ balancer://couch-slave$1 [P]
```

The following ProxyPassReverse directives instructs Apache to adjust the URLs in the HTTP response headers to match that of the proxy server, instead of the reverse proxied server. This is mainly useful for the Location header that is sent when CouchDB creates a new document:

```
ProxyPassReverse /api http://couch-master:5984/api
ProxyPassReverse /api balancer://couch-slave
```

Open */etc/apache2/apache2.conf* and look for the ServerLimit, ThreadsPerChild, and MaxClients directives. Apache limits the MaxClients to the ServerLimit multiplied by

the `ThreadsPerChild`. These directives are intended to prevent your server from running out of memory and swapping, which would significantly decrease performance. Following is an example configuration with the `MaxClients` increased to 5,000 (this is from a machine with 1 GB of RAM):

```
ServerLimit         200
ThreadsPerChild      25
MaxClients         5000
```

On `couch-proxy.example.com`, restart Apache:

```
sudo /etc/init.d/apache2 restart
```

Testing

Test your load balancer by making an HTTP request to the proxy server:

```
curl -X GET http://couch-proxy.example.com/api
```

It should proxy the request through to one of the CouchDB nodes and respond as follows (your details will be different):

```
{
   "db_name":"api",
   "doc_count":0,
   "doc_del_count":0,
   "update_seq":0,
   "purge_seq":0,
   "compact_running":false,
   "disk_size":79,
   "instance_start_time":"1296720556231838",
   "disk_format_version":5,
   "committed_update_seq":0
}
```

Let's try and `POST` a new document to the load balancer, treating it as if it's a CouchDB node:

```
curl -X POST http://couch-proxy.example.com/api \
-H "Content-Type: application/json" \
-d '{
   "_id":"doc-a"
}'
```

The response:

```
{
   "ok":true,
   "id":"doc-a",
   "rev":"1-967a00dff5e02add41819138abb3284d"
}
```

Let's try and `GET` the newly created document from the load balancer, again treating it as if it's a CouchDB node:

```
curl -X GET http://couch-proxy.example.com/api/doc-a
```

The response:

```
{
   "_id":"doc-a",
   "_rev":"1-967a00dff5e02add41819138abb3284d"
}
```

If you `GET` the newly created document from `couch-a.example.com`, then you should get the exact same response:

```
curl -X GET http://couch-a.example.com:5984/api/doc-a
```

If you `GET` the newly created document from `couch-b.example.com`, then you should get the exact same response:

```
curl -X GET http://couch-b.example.com:5984/api/doc-a
```

Finally, if you `GET` the newly created document from `couch-c.example.com`, then you should get the exact same response:

```
curl -X GET http://couch-c.example.com:5984/api/doc-a
```

If the document did not replicate from one CouchDB node to the other, then make sure that continuous replication is running.

See Chapter 6 for information about how to perform distributed load testing. When load testing, watch Apache's error log for errors such as "server reached MaxClients setting, consider raising the MaxClients setting" or "server is within MinSpareThreads of MaxClients, consider raising the MaxClients setting," and adjust Apache's settings as described earlier.

After changing these settings and restarting Apache, watch for startup warnings such as "WARNING: MaxClients of 1000 would require 40 servers, and would exceed the ServerLimit value of 16. Automatically lowering MaxClients to 400. To increase, please see the ServerLimit directive." As mentioned before, Apache limits the `MaxClients` to the `ServerLimit` multiplied by the `ThreadsPerChild`.

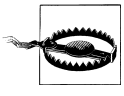
CHAPTER 5
Clustering

In Chapter 4, we looked at load balancing. Load balancing is very useful, but it alone may not provide you with the scale you need. Sometimes it is necessary to partition your data across multiple *shards*. Each shard lives on a CouchDB node and contains a subset of your data. You can have one or more shards on each node. CouchDB does not natively support this form of clustering. However, there are third-party tools that allow you to create a cluster of CouchDB nodes. These tools include BigCouch, Lounge, and Pillow.

An alternative to automatic partitioning is to manually partition your documents into different databases by type of document. The downside to this approach is that only documents in the same database can be included in any given view. If you have documents that don't need to be queried in the same view, putting them in separate databases can allow you to use CouchDB as-is without needing a third-party tool.

BigCouch

BigCouch is a fork of CouchDB that introduces additional clustering capabilities. It is available under an open source license and is maintained by Cloudant. For the most part, you can interact with a BigCouch node exactly the same way you would interact with a CouchDB node. BigCouch introduces some new API endpoints that are needed to manage its clustering features.



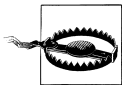
BigCouch is actively being developed. As of this writing, the current version of BigCouch was 0.3. Features and capabilities may change in future releases.

Each BigCouch node keeps a list of nodes that are part of its cluster. Each node is equally capable of handling requests, so you will want to load balance requests to your BigCouch nodes. Documents and views are partitioned by BigCouch across the shards

using a deterministic hashing algorithm. Read and write operations are done using quorum protocols. View results are created on each shard and then merged by the coordinating node when queried. The `_changes` feed is merged in a similar way, but without a global sort order and uses strings instead of sequence numbers. There are four parameters that control the operation of a BigCouch cluster. These parameters are described in Table 5-1.

Table 5-1. BigCouch Cluster Parameters

Parameter	Description
Q	The total number of shards, or partitions, to divide the documents in your database across. Note that multiple shards may exist on a single node, allowing you to grow the number of nodes in your cluster without needing to re-shard. This is configured in the *default.ini* file, but may be specified using the q query parameter when creating a database. The default value is 8.
N	The number of redundant replicas that should be created for each document (technically, the number of copies of each shard), on the same number of different nodes. The default value is 3.
W	The number needed for a write quorum. A client will not receive an indication of write success (`201 Created`) until this many nodes have successfully written the document. W must be less than or equal to N. If W is less than N, then the remaining writes will still be attempted in the background. This is configured in the *default.ini* file, but may be specified using the `w` query parameter when writing to the database. The default value is 2.
R	The number needed for a read quorum. A document will not be returned to a client until this many successful reads on different nodes have been made, all with the same revision number. R must be less than or equal to N. This is configured in the *default.ini* file, but may be specified using the `r` query parameter when reading from the database. The default value is 2.



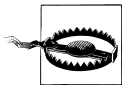
Each BigCouch node needs to agree on a magic cookie value. For security reasons, be sure to change this magic cookie value to a new value (using the same new value on each node) in each node's *vm.args* file.

Lounge

Lounge is another tool that allows you to create a cluster of CouchDB nodes. It is available under an open source license and is maintained by Meebo. Lounge takes a different approach to clustering than BigCouch. Lounge is itself a proxy server that manages a cluster of CouchDB nodes. Lounge requires a small patch to CouchDB that enables design-only replication (so that only design documents and views are replicated) but, other than that, uses CouchDB as-is.

Lounge actually includes two proxy servers—a *dumbproxy* and a *smartproxy*. The dumbproxy is nginx packaged up with a custom proxying module. The dumbproxy handles routing requests to the correct shard, or partition, for documents and for everything else that is not a view. Document IDs are hashed using a consistent hashing algorithm. This hash determines to which node an HTTP request gets sent. Since the document ID is included in both reads and writes, the dumbproxy will always send both reads and writes to the correct node. The smartproxy is a Python Twisted daemon

which handles the sharding of views—see Chapter 19 in CouchDB: The Definitive Guide (O'Reilly). Finally, Lounge includes a *replicator* which keeps design documents synchronized and can replicate documents for redundancy.



Neither BigCouch nor Lounge support automatic resharding. For this reason, you may want to consider oversharding your database. Basically, this means having multiple shards on each CouchDB node. For example, if you have 8 shards on 2 nodes, this gives you room to grow to 8 nodes. In this scenario, once you've reached 8 nodes you would not be able to automatically reshard your database. If you wanted to reshard to support 64 shards, this would be a manual process.

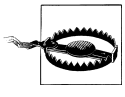
Pillow

Pillow is both a router and rereducer for CouchDB. Like Lounge, Pillow is a separate application that sits in front of a cluster of standard CouchDB nodes. Both Lounge and Pillow are designed to solve the same problems, but they have different approaches to solving these problems. Pillow uses the hash of a document ID to determine which CouchDB node should serve the request. For view requests, Pillow collects the view results from all CouchDB nodes and merges these results for the client.

One nice feature of Pillow is that it supports automatic resharding. Resharding is done using CouchDB's replicator. The shard to which to send an individual document is determined by replication filters. When asked to reshard, Pillow will create the necessary replication filters and initiate the replication. You can then monitor the replication status to see when Pillow is ready for you to switch to using the new shards. Pillow can automatically switch to the new shards when they are ready, but it is recommended that you make the switch manually.

CHAPTER 6
Distributed Load Testing

Each application has its own unique usage patterns. It can be very difficult to accurately predict these usage patterns. If you are working with an existing system, then you can take a look at log files and analytics data to get a sense of how your application is used. If this is a new system, then you can create scenarios based on how you expect the application to be used. Generic benchmarking can be of some use, but a test specifically designed for your system will be more useful.



The example load test in this chapter is intended as an illustrative example that can be helpful when you are writing your own tests. However, writing a test that is customized to your application's usage patterns can be very difficult. An appropriate test for your system will look very different than the example test in this chapter.

There are many tools available that allow you to create tests customized for your application. However, when creating a distributed system it can be difficult to actually generate enough load to push your system to its maximum capacity. In order to stress test a distributed system, you will need a distributed load testing tool. Tsung is a distributed load and stress testing tool that we will use for the example this chapter. We will be using Tsung on Ubuntu, but these steps can be easily adapted to other platforms. Tsung can generate `GET` and `POST` HTTP requests and, as of version 1.2.1, `PUT` and `DELETE` HTTP requests. Some of Tsung's features include:

- Monitoring of client operating systems' CPU, memory, and network traffic
- Simulation of dynamic sessions, described in an XML configuration file
- Randomized traffic patterns based on defined probabilities
- Recording of HTTP sessions for later playback during a test
- HTML reports and graphs

Like CouchDB, Tsung is developed in Erlang. Depending on the number of testing servers used, Tsung can simulate hundreds of thousands of concurrent users. Given enough servers, you could even simulate millions of concurrent users. In addition to being able to test HTTP servers, Tsung can also test WebDAV, SOAP, PostgreSQL, MySQL, LDAP, and Jabber/XMPP servers.

Installing Tsung

In the following examples, we will have two testing clients with domain names of `test-a.example.com` and `test-b.example.com`, and we will be testing our `couch-proxy.example.com` (load balancer), `couch-master.example.com` (CouchDB master node), `couch-a.example.com` (CouchDB read-only node), `couch-b.example.com` (CouchDB read-only node), `couch-c.example.com` (CouchDB read-only node) servers. Install Erlang on both `test-a.example.com` and `test-b.example.com`:

```
sudo aptitude install erlang
```

Install gnuplot on both `test-a.example.com` and `test-b.example.com`:

```
sudo aptitude install gnuplot
```

Install Perl's Template Toolkit on both `test-a.example.com` and `test-b.example.com`:

```
sudo aptitude install libtemplate-perl
```

Install Python's Matplotlib on both `test-a.example.com` and `test-b.example.com`:

```
sudo aptitude install python-matplotlib
```

Download the latest version of Tsung on both `test-a.example.com` and `test-b.example.com`. As of this writing, this was version 1.3.3:

```
wget http://tsung.erlang-projects.org/dist/tsung-1.3.3.tar.gz
```

Extract the downloaded file on both `test-a.example.com` and `test-b.example.com`:

```
tar -xzf tsung-1.3.3.tar.gz
```

On both `test-a.example.com` and `test-b.example.com`, change into the *tsung-1.3.3* directory:

```
cd tsung-1.3.3
```

Configure on both `test-a.example.com` and `test-b.example.com`:

```
sudo ./configure
```

Make on both `test-a.example.com` and `test-b.example.com`:

```
sudo make
```

Install on both `test-a.example.com` and `test-b.example.com`:

```
sudo make install
```

We will be launching our tests from `test-a.example.com`, so you will need to be able to login from this client to `test-b.example.com` without using a password. We will do this using public key authentication, but you could instead use ssh-agent or rsh. Install the OpenSSH server on `test-a.example.com` and `test-b.example.com`, if it is not already:

```
sudo aptitude install openssh-server
```

Each of your testing clients should use the same username for running tests. Setting up multiple machines will be much easier if the same username is used on each. If you use Erlang-based remote monitoring, create this same username on each server to be monitored as well. Alternatively, you can configure default usernames in your users' SSH configuration files.

Generate an SSH key on `test-a.example.com`, if you have not already:

```
ssh-keygen
```

Pick the default file in which to save the key, likely *~/.ssh/id_rsa*. Enter a passphrase, if you'd like.

From `test-a.example.com`, copy the *~/.ssh/id_rsa* public key to `test-b.example.com`:

```
scp ~/.ssh/id_rsa.pub test-b.example.com:~
```

If this is the first time using SSH to connect from `test-a.example.com` to `test-b.example.com`, you will need to accept the RSA key fingerprint. Enter the user's password and you should see output indicating that the file has been copied.

Log into `test-b.example.com` and add the public key copied from `test-a.example.com` to `test-b.example.com`'s list of authorized keys:

```
cat ~/id_rsa.pub >> ~/.ssh/authorized_keys
```

Still on `test-b.example.com`, remove the public key that was copied over from `test-a.example.com`:

```
rm ~/id_rsa.pub
```

To test, try logging into `test-b` from `test-a.example.com`, accepting the RSA key fingerprint if prompted, and you should not be prompted for a password:

```
ssh test-b
```

We are using the local hostname, `test-b`, rather than the complete hostname, `test-b.example.com`, since Tsung will require us to use the local hostname in our test configuration. Tsung will also allow you to use the IP address of the machine, if you'd prefer.

If you have additional testing clients, repeat the above steps for each, setting up `test-a.example.com` to be able to log into each testing machine without using a password. The more testing servers you have, the more simulated load you can generate.

Oddly enough, `test-a.example.com` will also need to be able to log into itself without using a password. To add its own public key to its list of authorized keys, from `test-a.example.com`:

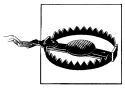
```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

To test, try logging into `test-a` from `test-a.example.com` (yes, from itself), accepting the RSA key fingerprint if prompted, and you should not be prompted for a password:

```
ssh test-a
```

Configuring Tsung

Tsung comes with an example configuration file for doing distributed HTTP testing, which you'll find in */usr/share/doc/tsung/examples/http_distributed.xml*. We will create our own configuration file, saved to *~/http_distributed_couch_proxy.xml* (see the Tsung User's manual at *http://tsung.erlang-projects.org/user_manual.html*):



The location of the DTD file, */usr/share/tsung/tsung-1.0.dtd*, may be different on your testing client. If so, this value will need to be modified to match the location on your client.

```
<?xml version="1.0"?>
<!DOCTYPE tsung SYSTEM "/usr/share/tsung/tsung-1.0.dtd">
<tsung loglevel="notice" version="1.0">

  <!-- Client side setup -->
  <clients>
    <client host="test-a" weight="1" maxusers="10000" cpu="4"/>
    <client host="test-b" weight="1" maxusers="10000" cpu="4"/>
  </clients>

  <!-- Server side setup -->
  <servers>
    <server host="couch-proxy" port="80" type="tcp"/>
  </servers>

  <!-- Load setup -->
  <load>
    <arrivalphase phase="1" duration="5" unit="minute">
      <users arrivalrate="200" unit="second"></users>
    </arrivalphase>
  </load>

  <!-- Sessions setup -->
  <sessions>
```

```
<session name="post_get" probability="2.5" type="ts_http">
  <thinktime value="10" random="true"/>
  <setdynvars sourcetype="random_number" start="2008" end="2011">
    <var name="yyyy"/>
  </setdynvars>
  <setdynvars sourcetype="random_number" start="10" end="12">
    <var name="mm"/>
  </setdynvars>
  <setdynvars sourcetype="random_number" start="10" end="28">
    <var name="dd"/>
  </setdynvars>
  <request subst="true">
    <match do="abort" when="nomatch">201 Created</match>
    <dyn_variable name="id" jsonpath="$.id"/>
    <dyn_variable name="rev" jsonpath="$.rev"/>
    <http
      method="POST"
      url="/api"
      content_type="application/json"
      contents="{
        &amp;quot;date&amp;quot;:[
          &amp;quot;%%_yyyy%%&amp;quot;,
          &amp;quot;%%_mm%%&amp;quot;,
          &amp;quot;%%_dd%%&amp;quot;            ]
      }"
    >
      <http_header name="Accept" value="application/json"/>
    </http>
  </request>
  <for from="0" to="9" incr="1" var="x">
    <thinktime value="10" random="true"/>
    <request subst="true">
      <match do="abort" when="nomatch">304 Not Modified</match>
      <http method="GET" url="/api/%%_id%%">
        <http_header name="If-None-Match" value="&amp;quot;%%_rev%%&amp;quot;"/>
        <http_header name="Accept" value="application/json"/>
      </http>
    </request>
  </for>
</session>

<session name="put_get" probability="2.5" type="ts_http">
  <thinktime value="10" random="true"/>
  <setdynvars sourcetype="random_string" length="32">
    <var name="id"/>
  </setdynvars>
  <setdynvars sourcetype="random_number" start="2008" end="2011">
    <var name="yyyy"/>
  </setdynvars>
  <setdynvars sourcetype="random_number" start="10" end="12">
    <var name="mm"/>
  </setdynvars>
  <setdynvars sourcetype="random_number" start="10" end="28">
    <var name="dd"/>
  </setdynvars>
```

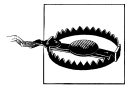
```
      <request subst="true">
        <match do="abort" when="nomatch">201 Created</match>
        <dyn_variable name="rev" jsonpath="$.rev"/>
        <http
          method="PUT"
          url="/api/%%_id%%"
          content_type="application/json"
          contents="{
            &quot;date&quot;:[
              &quot;%%_yyyy%%&quot;,
              &quot;%%_mm%%&quot;,
              &quot;%%_dd%%&quot;
            ]
          }"
        >
          <http_header name="Accept" value="application/json"/>
        </http>
      </request>
      <for from="0" to="9" incr="1" var="x">
        <thinktime value="10" random="true"/>
        <request subst="true">
          <match do="abort" when="nomatch">304 Not Modified</match>
          <dyn_variable name="rev" jsonpath="$._rev"/>
          <http method="GET" url="/api/%%_id%%">
            <http_header name="If-None-Match" value="&quot;%%_rev%%&quot;"/>
            <http_header name="Accept" value="application/json"/>
          </http>
        </request>
      </for>
    </session>

    <session name="view_pagination" probability="20" type="ts_http">
      <thinktime value="10" random="true"/>
      <request subst="true">
        <http
          method="GET"
          url="/api/_design/default/_view/dates?reduce=false&amp;skip=0&amp;limit=10"
        >
          <http_header name="Accept" value="application/json"/>
        </http>
      </request>
      <for from="10" to="90" incr="10" var="skip">
        <thinktime value="10" random="true"/>
        <request subst="true">
          <http
            method="GET"
            url="/api/_design/default/_view/dates?reduce=false&amp;skip=%%_skip%%
&amp;limit=10&amp;stale=ok"
          >
            <http_header name="Accept" value="application/json"/>
          </http>
        </request>
      </for>
    </session>
```

```
    <session name="view_grouped" probability="75" type="ts_http">
      <thinktime value="10" random="true"/>
      <request>
        <http
          method="GET"
          url="/api/_design/default/_view/dates?group_level=1"
        >
          <http_header name="Accept" value="application/json"/>
        </http>
      </request>
      <thinktime value="10" random="true"/>
      <request>
        <http
          method="GET"
          url="/api/_design/default/_view/dates?group_level=2&amp;stale=ok"
        >
          <http_header name="Accept" value="application/json"/>
        </http>
      </request>
    </session>

  </sessions>

</tsung>
```



All client hosts must be able to resolve the hostname of the machine running the tests. In the above configuration, `test-b` must be able to resolve the IP address of `test-a`.

Let's walk through some parts of this configuration file. First, the `clients` element:

```
<!-- Client side setup -->
<clients>
  <client host="test-a" weight="1" maxusers="10000" cpu="4"/>
  <client host="test-b" weight="1" maxusers="10000" cpu="4"/>
</clients>
```

This `clients` element contains a list of clients from which tests may be launched. The more clients, the greater the simulated load that can be generated. Each client needs to be configured using its local hostname or IP address using the `host` attribute. The `weight` attribute assigns a relative weight to the client since some clients may be faster and able to start more sessions than other clients. The `maxusers` attribute defines a maximum number of users to simulate on this client. The `cpu` attribute declares how many Erlang virtual machines Tsung should use and should be the same as the number of CPUs that are available to the client.

The `servers` element:

```
<!-- Server side setup -->
<servers>
  <server host="couch-proxy" port="80" type="tcp"/>
</servers>
```

The `servers` element contains a list of servers to be tested. Each server needs to be configured using its local hostname or IP address using the `host` attribute. The `port` attribute indicates the TCP/IP port number to use. The type attribute can either be `tcp` or `udp`. Since HTTP uses TCP, we're using `tcp` as the value here.

The `load` element:

```
<!-- Load setup -->
<load>
  <arrivalphase phase="1" duration="5" unit="minute">
    <users arrivalrate="200" unit="second"></users>
  </arrivalphase>
</load>
```

The `load` element contains a list of `arrivalphase` elements, each simulating various types of load. The `arrivalphase` element's `phase` attribute represents the sequential number of the arrival phase. Here we are only defining one arrival phase. The `duration` attribute defines how long the arrival phase should last and the `unit` attribute defines the unit by which to measure the duration. Possible values for the `unit` element are `second`, `minute`, or `hour`.

Within the `arrivalphase` element is a `users` element. The `arrivalrate` attribute of the `users` element defines the number of arrivals within the timeframe defined by the `unit` element. Possible values for the `unit` element are `second`, `minute`, or `hour`. Here we are telling Tsung to start 200 "arrivals" every second for 5 minutes.

The `sessions` element:

```
<!-- Sessions setup -->
<sessions>
  …
</sessions>
```

The `sessions` element contains a list of `session` elements. These each represent user sessions which may be simulated. You can define multiple sessions and each can have its own probability, but the total probability of all sessions must add up to 100. Let's take a look at each session individually.

The `session` element with the `name` attribute value of `post_get`:

```
<session name="post_get" probability="2.5" type="ts_http">
  …
</session>
```

This `session` element contains a `name` attribute with the value of `post_get`. This name will be used in reports to identify the session. The `session` element's `probability` attribute indicates the percent probability of this session being used for any given user. Remember, the total probability of all sessions must add up to 100. The `session` element's type attribute can be either `ts_http`, `ts_jabber`, or `ts_mysql`. Since we're using HTTP, the type is `ts_http`.

The `thinktime` element:

```
<thinktime value="10" random="true"/>
```

The `thinktime` element defines an amount of time to wait, or "think", before continuing. This is helpful when trying to more realistically simulate load. The `thinktime` element's `value` attribute is the amount of time, in seconds, to wait. Setting the `thinktime` element's `random` attribute to a value of `true` tells Tsung to randomize the wait time, using the `value` attribute's value as the mean.

The `setdynvars` elements:

```
<setdynvars sourcetype="random_number" start="2008" end="2011">
  <var name="yyyy"/>
</setdynvars>
<setdynvars sourcetype="random_number" start="10" end="12">
  <var name="mm"/>
</setdynvars>
<setdynvars sourcetype="random_number" start="10" end="28">
  <var name="dd"/>
</setdynvars>
```

Each of the `setdynvars` elements sets a dynamic variable. The `sourcetype` attribute value of `random_number` tells Tsung to generate a random number. The `start` and `end` attributes indicate the starting and ending values, respectively, to use when generating the random number. The nested `var` element actually instantiates the variable, using the variable name defined in the `name` attribute. Here we are generating random year, month, and day values which we will use later in the session.

A `request` element:

```
<request subst="true">
  <match do="abort" when="nomatch">201 Created</match>
  <dyn_variable name="id" jsonpath="$.id"/>
  <dyn_variable name="rev" jsonpath="$.rev"/>
  <http
    method="POST"
    url="/api"
    content_type="application/json"
    contents="{
      &amp;quot;date&amp;quot;:[
        &amp;quot;%%_yyyy%%&amp;quot;,
        &amp;quot;%%_mm%%&amp;quot;,
        &amp;quot;%%_dd%%&amp;quot;
      ]
    }"
  >
    <http_header name="Accept" value="application/json"/>
  </http>
</request>
```

A `request` element defines a request to be made as part of the session. Since we'll be using the dynamic variables defined earlier, we need set the `request` element's `subst` attribute's value to `true`. This tells Tsung to substitute variables for their values, when encountered.

The `match` element tells Tsung to "match" on a certain condition. The `do` attribute value of `abort` tells Tsung to abort the session if the match condition is true. Possible values for the `do` attribute are `continue`, `log`, `abort`, `restart`, or `loop`. The `when` attribute can either be `match` or `nomatch`. The text of the `match` element is the text to match or not match on. In this case, if the text `201 Created` is not found in the response (i.e., the document was not created) then we abort the session.

The two `dyn_variable` elements define dynamic variables that will be based on the server's response. The `name` attribute defines the name of the variable to use. Tsung allows matching using a limited subset of JSONPath (XPath for JSON), using the `jsonpath` attribute. These two variables will contain the ID and revision of the created document (once the response has been received).

The `http` element initiates an HTTP request. The `method` attribute specifies the HTTP method to use for the request (e.g., `GET`, `POST`, `PUT`, `DELETE`). The `url` attribute specifies the URL to which to make the request. This can be relative to the host set up earlier in the `servers` element, or a full URL. The `content_type` attribute specifies the value of the `Content-Type` HTTP header. The contents attribute specifies the contents of a `POST` or `PUT` request body. Here we are using a JSON object as the request body. The JSON object contains one field, `date`, with its value being an array of year, month, and day values (using the random dynamic variables created earlier).

A `for` element:

```
<for from="0" to="9" incr="1" var="x">
  <thinktime value="10" random="true"/>
  <request subst="true">
    <match do="abort" when="nomatch">304 Not Modified</match>
    <http method="GET" url="/api/%%_id%%">
      <http_header name="If-None-Match" value="&quot;%%_rev%%&quot;"/>
      <http_header name="Accept" value="application/json"/>
    </http>
  </request>
</for>
```

A `for` element will tell Tsung to repeat the enclosed directives a specified number of times. Here we are using a `from` value of `0`, a `to` value of `9`, an `incr` value of `1`, and using a `var` (variable) with a name of `x`. This means that the variable `x` will start out with the value of `0`, increment by `1` in each iteration, and the loop will stop when `x` has reached the value of `9`.

The contained `thinktime`, `request`, `match`, and `http` elements should look familiar. Within the `http` element, you'll see two `http_header` elements. As you may have guessed, these specify the `name` and `value` of HTTP headers to send as part of the request.

The `If-None-Match` HTTP header allows us to use conditional caching and the Accept header tells CouchDB that our client can handle content of type `application/json`.

The remaining sessions in the configuration file should be self-explanatory.

Running Tsung

First, we need to create the view that is used in the above configuration file. This is simply a view of dates (as an array of year, month, and day) from our documents:

```
curl -X PUT http://couch-proxy.example.com/api/_design/default -d \
'{
   "language": "javascript",
   "views": {
       "dates": {
           "map":
"function(doc) {
  if (doc.date) {
    emit(doc.date);
  }
}",
           "reduce": "_count"
       }
   }
}'
```

The response:

```
{
   "ok":true,
   "id":"_design/default",
   "rev":"1-edb41165ec8e4839dd7918e88e2125fa"
}
```

Start Tsung, telling it to use the above configuration file:

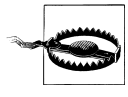
```
tsung -f ~/http_distributed_couch_proxy.xml start
```

Note that Tsung will wait for all sessions to complete before finishing, even if it takes longer than the duration of all phases. Tsung will let you know what directory it has logged to, for example:

```
"Log directory is: /home/bradley-holt/.tsung/log/20110221-23:26"
```

Change into the log directory and generate the HTML and graph reports using the *tsung_stats.pl* script package with Tsung:

```
/usr/lib/tsung/bin/tsung_stats.pl
```



The location of the *tsung_stats.pl* script may be different on your testing client. If so, you will need to locate this script and run it from the appropriate location.

If everything works correctly, a `report.html` file will be created in this same directory. Open this report and you will see several statistics and graphs. Under the statistics reports, the main statistics table shows you the highest 10 second mean, lowest 10 second mean, highest rate, mean, and count for each part of the HTTP connection. See Table 6-1 for a sample main statistics report.

Table 6-1. Main statistics

Name	Highest 10sec mean	Lowest 10sec mean	Highest rate	Mean	Count
connect	0.27 sec	0.663 msec	563.3 / sec	10.75 msec	161433
page	1.15 sec	3.91 msec	827.4 / sec	0.26 sec	229212
request	1.15 sec	3.91 msec	827.4 / sec	0.26 sec	229212
session	3min 29sec	6.41 sec	206.1 / sec	41.41 sec	56619

Tsung allows you to group requests into transactions. A transaction might be useful when testing an HTML page as you could group the requests for the HTML and all related assets (e.g., JavaScript, CSS, and images) into one transaction. We have not done this here, so your transactions statistics table will be empty. The network throughput table lets you see the size of the network traffic received and sent. See Table 6-2 for a sample network throughput report.

Table 6-2. Network throughput

Name	Highest rate	Total
size_rcv	3.60 Mbits/sec	126.19 MB
size_sent	1.01 Mbits/sec	35.83 MB

The counter statistics reports let you see the total number of simulated users and related statistics. See Tables 6-3 and 6-4 for sample counter statistics reports.

Table 6-3. Counter statistics, first table

Name	Highest rate	Total number
finish_users_count	206.1 / sec	56619
match	109.8 / sec	31041
match_stop	0.3 / sec	11
nomatch	0.3 / sec	11
users_count	196.3 / sec	56620

Table 6-4. Counter statistics, second table

Name	Max
connected	2797
users	8191

The HTTP return code table gives you a list of return codes with the highest rate and total number for each. See Table 6-5 for a sample HTTP return code report. In this sample report, we can see that we got the `200 OK` response code 198,157 times, and the highest rate for that response code was 713.1 / sec.

Table 6-5. HTTP return code

Code	Highest rate	Total number
200	713.1 / sec	198157
201	11.5 / sec	2832
304	100.2 / sec	28209
404	0.3 / sec	11

There are also several graphs reports available. For all graphs, the x-axis represents a progression of time throughout the test. The first graph represents mean transaction response time. The y-axis for this graph represents the mean number of milliseconds that the transaction response took during a given moment in the test. See Figure 6-1 for an example of this graph.

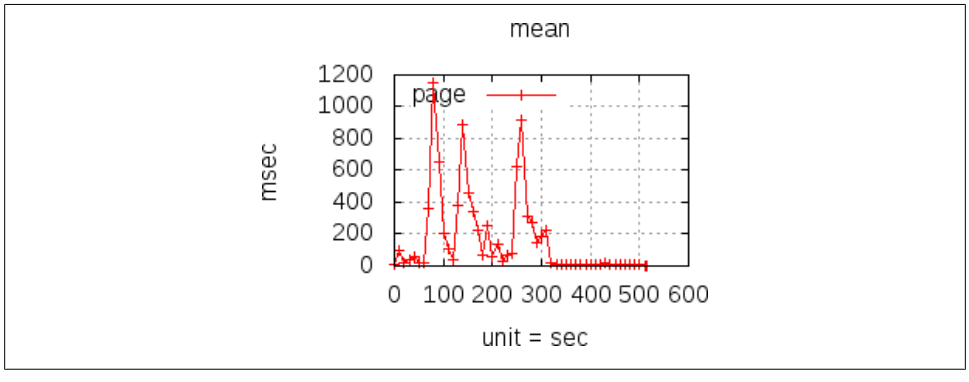


Figure 6-1. Graph of mean transaction response time

For the next graph, the y-axis represents the mean number of milliseconds that the request or connection establishment took during a given moment in the test. See Figure 6-2 for an example of this graph.

For the next graph, the y-axis represents the number of transactions per second during a given moment in the test. See Figure 6-3 for an example of this graph.

For the next graph, the y-axis represents the number of requests or connects during a given moment in the test. See Figure 6-4 for an example of this graph.

For the next graph, the y-axis represents the number of kilobits per second sent or received during a given moment in the test. See Figure 6-5 for an example of this graph.

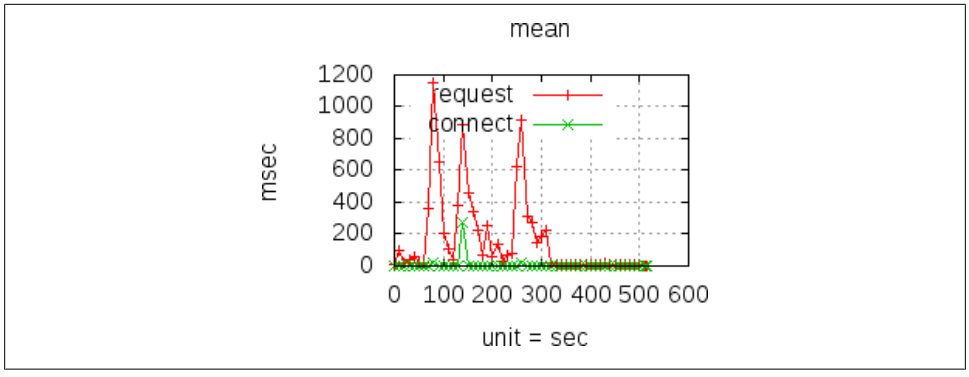


Figure 6-2. Graph of mean request and connection establishment response time

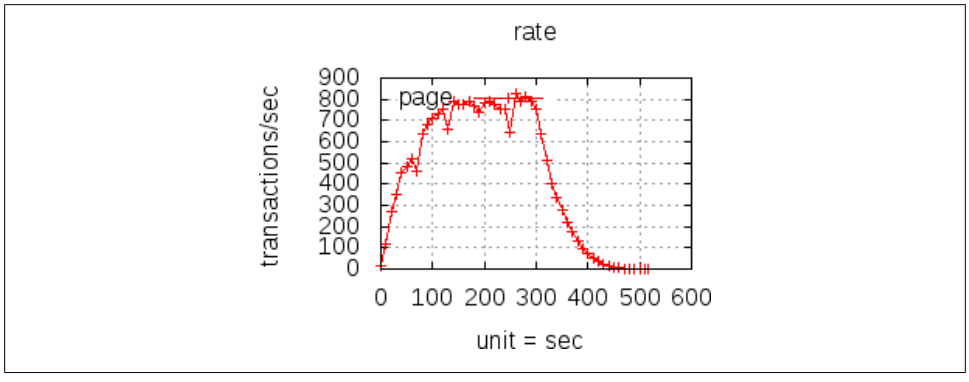


Figure 6-3. Graph of transactions rate throughput

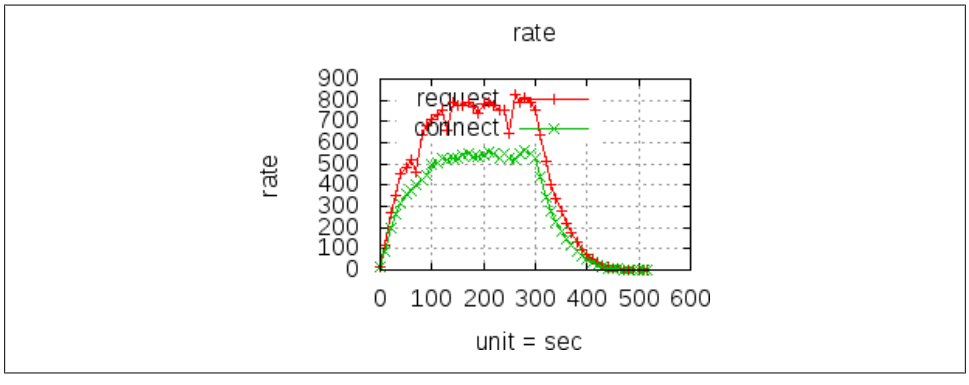


Figure 6-4. Graph of requests rate throughput

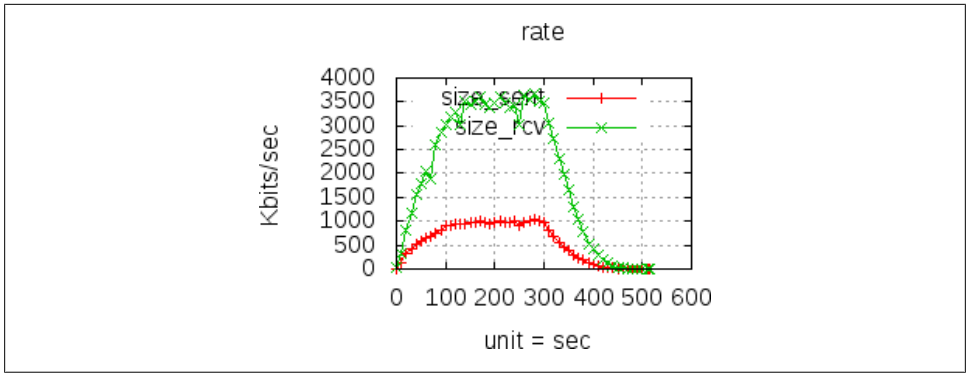


Figure 6-5. Graph of network traffic throughput

For the next graph, the y-axis represents the number of users per second arriving or finishing during a given moment in the test. See Figure 6-6 for an example of this graph.

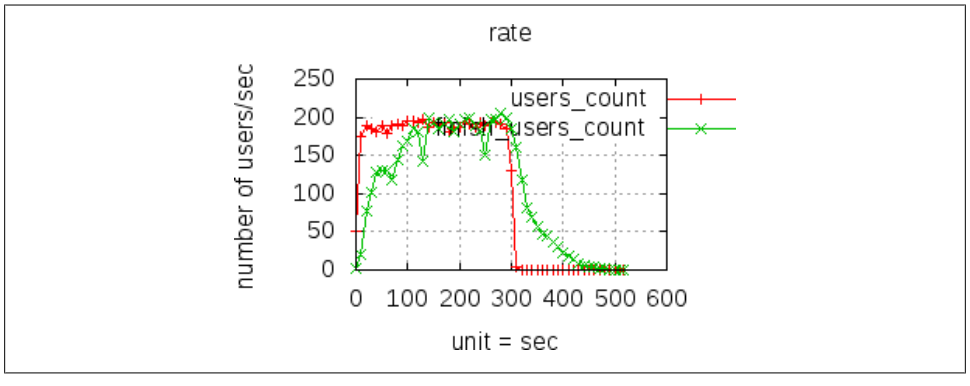


Figure 6-6. Graph of new users arrival rate throughput

For the next graph, the y-axis represents the number of simultaneous users arriving or connected during a given moment in the test. See Figure 6-7 for an example of this graph.

For the final graph, the y-axis represents the number of responses per second of a given HTTP return code. See Figure 6-8 for an example of this graph.

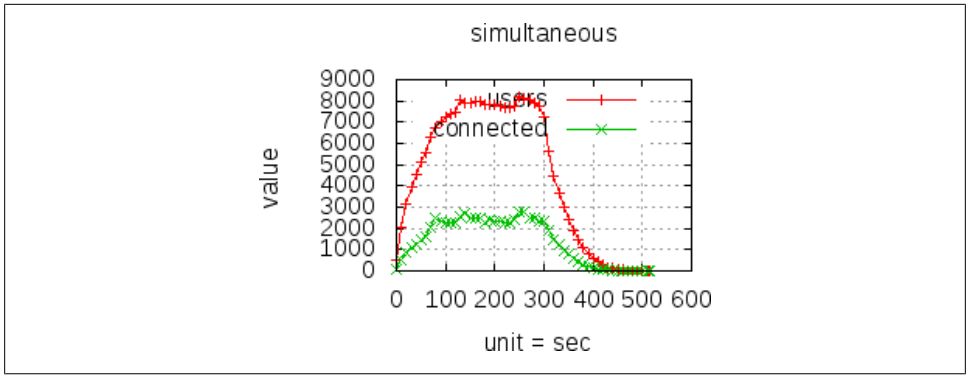


Figure 6-7. Graph of simultaneous users

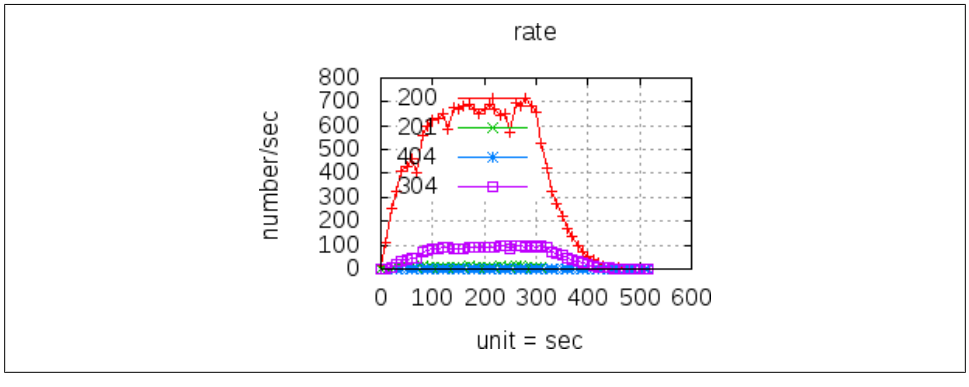


Figure 6-8. Graph of HTTP return code status rate

Monitoring

Tsung can monitor the CPU usage, memory usage, and load on your servers as they are being tested. This data can be used to better allocate your available network resources. You can use either Erlang, SNMP, or Munin to monitor your servers. In this example, we will use Munin for monitoring. This requires Munin to be installed on your servers. Install Munin on `couch-proxy.example.com`, `couch-master.example.com`, `couch-a.example.com`, `couch-b.example.com`, and `couch-c.example.com`:

```
sudo aptitude install munin-node
```

Next, you need to configure all of your Munin nodes to allow monitoring by `test-a.example.com`. Edit */etc/munin/munin-node.conf* and add the following line, replacing the IP address quadrants in `^10\.179\.113\.200$` with the quadrants from the IP address of your `test-a.example.com` testing server (as you may have guessed, the value of the `allow` directive is a regular expression):

```
allow ^10\.179\.113\.200$
```

To enable monitoring, add a `<monitoring>` section to *~/http_distributed_couch_proxy.xml* after your server side setup:

```
<!-- Monitoring setup -->
<monitoring>
  <monitor host="couch-proxy" type="munin"></monitor>
  <monitor host="couch-master" type="munin"></monitor>
  <monitor host="couch-a" type="munin"></monitor>
  <monitor host="couch-b" type="munin"></monitor>
  <monitor host="couch-c" type="munin"></monitor>
</monitoring>
```

Start Tsung, telling it to use the above configuration file, as before:

```
tsung -f ~/http_distributed_couch_proxy.xml start
```

As before, Tsung will let you know what directory it has logged to:

```
"Log directory is: /home/bradley-holt/.tsung/log/20110221-23:39"
```

Just like before, change into the log directory and generate the HTML and graph reports using the *tsung_stats.pl* script:

```
/usr/lib/tsung/bin/tsung_stats.pl
```

Open the `report.html` file. This time you will see a few new statistics and graphs. On the statistics report page, under server monitoring, you will see several statistics from the monitored servers. See Table 6-6 for an example of the server monitoring report. See Table 6-7 for a description of each measurement.

Table 6-6. Server monitoring

Name	Highest 10sec mean	Lowest 10sec mean
cpu:os_mon@couch-a	81.12 %	0.45 %
cpu:os_mon@couch-b	62.40 %	0.32 %
cpu:os_mon@couch-c	61.80 %	0.55 %
cpu:os_mon@couch-master	8.85 %	0.35 %
cpu:os_mon@couch-proxy	21.62 %	0.32 %
freemem:os_mon@couch-a	142.41 MB	100.57 MB
freemem:os_mon@couch-b	150.34 MB	116.84 MB
freemem:os_mon@couch-c	118.29 MB	98.40 MB
freemem:os_mon@couch-master	184.82 MB	179.22 MB
freemem:os_mon@couch-proxy	925.48 MB	450.45 MB
load:os_mon@couch-a	1.04	0.27
load:os_mon@couch-b	0.83	0.29
load:os_mon@couch-c	1.02	0.39
load:os_mon@couch-master	0.31	0.16
load:os_mon@couch-proxy	34.45	1.49

Table 6-7. Server monitoring measurements

Measure	Description
cpu	CPU utilization
freemem	Amount of free memory available
load	Server load

Under graphs reports, you will see a new server OS monitoring section. For the first graph, the y-axis represents the CPU utilization of your servers during a given moment in the test. See Figure 6-9 for an example of this graph.

For the next graph, the y-axis represents the free memory available on your servers during a given moment in the test. See Figure 6-10 for an example of this graph.

For the next graph, the y-axis represents the CPU load on your servers during a given moment in the test. See Figure 6-11 for an example of this graph.

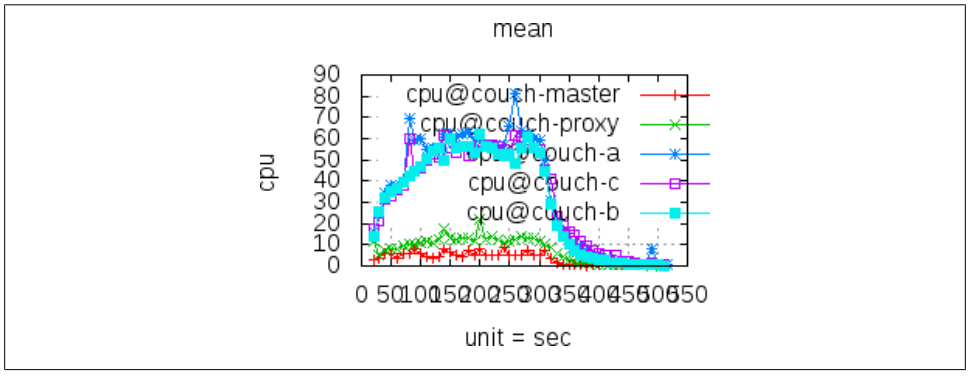


Figure 6-9. Graph of CPU utilization

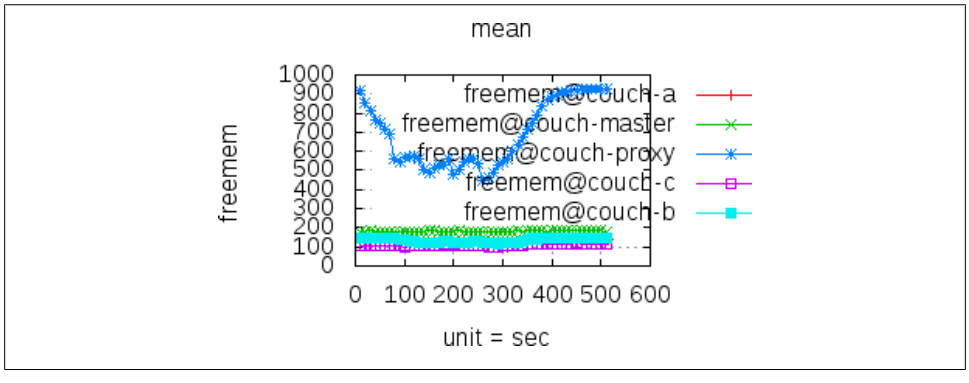


Figure 6-10. Graph of free memory

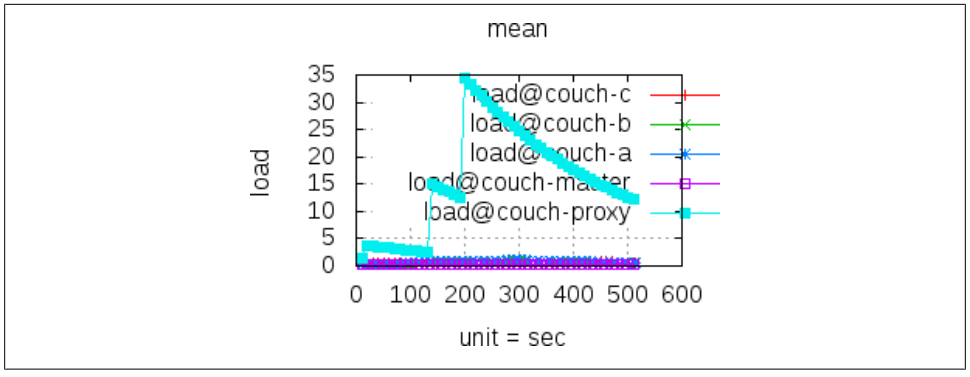


Figure 6-11. Graph of CPU load

Identifying Bottlenecks

Based on the results of the above tests, we can attempt to make a few conclusions. First, some analysis:

- The CPU utilization percentages on the read-only slave nodes, the write-only master node, and the proxy server are all quite low. It appears that none of these nodes are CPU bound.
- The free memory amounts on the read-only slave nodes, the write-only master node, and the proxy server never drops critically low. It looks like none of the nodes ever run out of memory, so excessive swapping should not be an issue.
- The server load averages on the read-only slave nodes and the write-only master node are reasonable.
- The server load average on the proxy server is quite high.

Based on this analysis, we might conclude that the proxy server is a potential bottleneck in our system. If you look back at the counter statistics, you'll see that the maximum number of connections reached was 2797. However, the maximum number of connections allowed to each read-only node was 4. With three read-only nodes, this gives us a total of 12 maximum connections to the backend CouchDB nodes. The write-only node did not have a limit, but our test scenarios were read-heavy. It appears that the proxy server is effectively queuing requests for the backend CouchDB nodes, which could account for the high server load.

Based on the above hypothesis, adding *more* read-only CouchDB nodes might actually lessen the load on the proxy server. Of course, we should test this hypothesis before we assume its validity. We're not going to do that here, but the point is that you should always challenge your assumptions with an actual test.

Test Configuration

The load tests in this chapter were performed using Rackspace Cloud Servers™. Each virtual server was on a machine with a Quad-Core AMD Opteron™ Processor running at 2200 MHz. The proxy server (`couch-proxy.example.com`) had 1024 MB of RAM and all other servers had 256 MB of RAM. The storage on all machines were 7200 RPM disks in a RAID-10 configuration (Rackspace would not disclose any more details about the hard drives). All machines were running Ubuntu 10.10 LTS (Maverick Meerkat), 64-bit. The configuration being tested matches that described in Chapter 4.