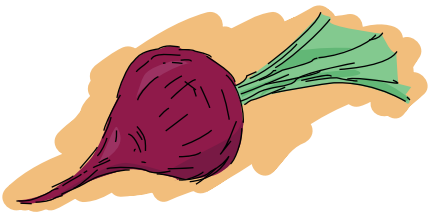
25

BEARS, ETS, BEETS: IN-MEMORY NOSQL FOR FREE!

Something we've been doing time and time again has been implementing some kind of storage device as a process. We've made fridges to store things, built `regis` to register processes, seen key/value stores, and so on. If we were programmers doing object-oriented design, we would have a bunch of singletons floating around, special storage classes, and whatnot. In fact, wrapping data structures like dicts and GB trees in processes is a bit like that.

This chapter introduces ETS, an in-memory database that provides an alternative data storage approach.

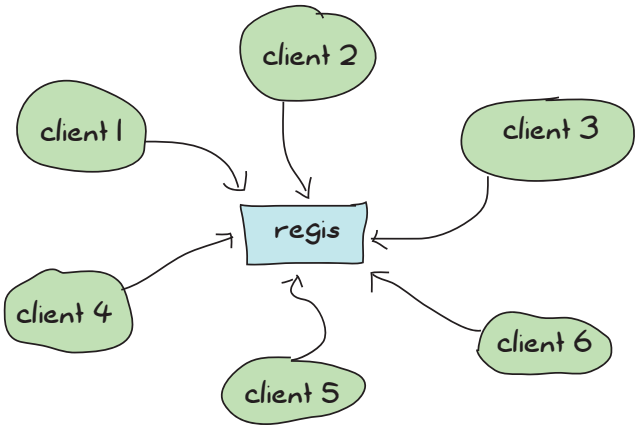


Why ETS

Holding data structures in a process is actually fine for a lot of cases, such as when you need that data to do some task within the process, as internal state, and so on. This will be the majority of our use cases. There is one case where it may not be the best choice: when the process holds a data structure for the sake of sharing it with other processes and little more.

One of the applications we've written is guilty of that. Can you guess which one? Of course you can. I mentioned it at the end of the previous chapter. `regis` (part of the Process Quest game we developed in Chapter 22) needs to be rewritten. That is not because it doesn't work or can't do its job well, but because it acts as a gateway to share data with potentially a lot of other processes.

`regis` is the central application to do messaging in Process Quest (and anything else that would use it), and pretty much every message going to a named process must go through it. This means that even though we took great care to make our applications very concurrent with independent actors and made sure our supervision structure could be scaled up, all of our operations will depend on a central `regis` process that will need to answer messages one by one:



If we have a lot of message-passing going on, `regis` risks getting busier and busier. If the demand is high enough, our whole system will become sequential and slow. That's pretty bad.

NOTE *We have no direct proof that `regis` is a bottleneck within Process Quest. In fact, Process Quest does very little messaging compared to many other applications in the wild. If we were using `regis` for something that required a lot more messaging and lookups, then the problems would be more apparent.*

To get around this problem, we could either split `regis` into subprocesses to make lookups faster (*sharding* the data) or find a way to store the data in some database that will allow for parallel and concurrent access of the data. While the first way would be very interesting to explore, we'll take an easier path by doing the latter.

Erlang has Erlang Term Storage (ETS) tables, which are an efficient in-memory database included with the Erlang VM. This database sits in a part of the VM where destructive updates are allowed and where garbage collection dares not approach, in a part of memory not shared by processes. ETS tables are generally fast, and they provide a pretty easy way for Erlang programmers to optimize some of their code when parts of it get too slow.

ETS tables allow limited concurrency in reads and writes (much better than none at all for a process's mailbox) in a way that could let us optimize away a lot of the pain, but that could also add problems. This is because their use throws away most of the concepts that make Erlang safe for concurrency in the first place.

> **DON'T DRINK TOO MUCH KOOL-AID**
>
> While ETS tables are a nice way to optimize applications, they should be used with some care. By default, the VM is limited to 1,400 ETS tables. While it is possible to change that number (by using `erl -env ERL_MAX_ETS_TABLES Number`), this default low level is a good sign that you should try to avoid having one table per process in general.

But before we rewrite `regis` to use ETS, we should try to understand a bit of ETS's principles.

The Concepts of ETS

ETS tables are implemented as BIFs in the `ets` module. ETS was designed to provide a way to store large amounts of data in Erlang with constant access time (functional data structures tend to flirt with logarithmic access time), and to have such storage look as if it were implemented as processes in order to keep their use simple and idiomatic.

NOTE *Having tables look like processes doesn't mean that you can spawn them or link to them. It means that they can respect semantics of nothing shared, wrapping calls behind functional interfaces, having them handle any native data type for Erlang, and making it possible to give them names (in a separate registry). Also, while you can't link to tables, they do have a similar mechanism available, as mentioned near the end of this section.*

All ETS tables natively store Erlang tuples, and only tuples. The tuples can contain whatever you want, and one of the tuple elements will act as

a primary key that you use to sort things. For example, having tuples of people of the form {Name, Age, PhoneNumber, Email} will let you have a table that looks like this:

```
{Name, Age, PhoneNumber, Email},
{Name, Age, PhoneNumber, Email},
{Name, Age, PhoneNumber, Email},
{Name, Age, PhoneNumber, Email},
{Name, Age, PhoneNumber, Email},
...
```

So if we want to have the table's index be the email addresses, we can do this by telling ETS to set the key position to 4 (as described in "Creating and Deleting Tables" on page 423). Once you've decided on a key, you can choose different ways to store data in tables:

`set`
: A `set` table will tell you that each key value must be unique. There can be no duplicate email in the preceding database example. Sets are great when you need to use a standard key/value store with constant time access.

`ordered_set`
: There can still be only one key instance per table, but `ordered_set` adds a few other interesting properties. The first is that elements in an `ordered_set` table will be ordered (who would have thought?). The first element of the table is the smallest one, and the last element is the largest one. If you traverse a table iteratively (jumping to the next element over and over again), the values should be increasing, which is not necessarily true of `set` tables. Using `ordered_set` tables is great when you frequently need to operate on ranges (for example, "I want entries 12 to 50"). They do, however, have the downside of being slower in their access time (`O(log N)`, where *N* is the number of objects stored).

`bag`
: A `bag` table can have multiple entries with the same key, as long as the tuples themselves are different. This means that the table can contain {key, some, values} and {key, other, values} without any problems, which would be impossible with `set` tables (they have the same key). However, you couldn't have {key, some, values} twice in the table, as those entries would be entirely identical.

`duplicate_bag`
: The tables of this type work like `bag` tables, except that they allow entirely identical tuples to be held multiple times within the same table.

NOTE *The `ordered_set` tables will see the values 1 and 1.0 as identical for all operations. Other tables will see them as different. Although it's nice to be able to use both integers and floating-point numbers together, it is somewhat rare that a given function can return both. You will generally avoid a lot of problems if your programs restrict their return values to one or the other.*

Another general concept related to ETS tables involves table ownership. When a process calls a function that starts a new ETS table, that process is the owner of the table.

By default, only the owner of the table can write to it, but everyone can read from it. This is known as the `protected` level of permissions. You can also choose to set the permissions to `public`, where everyone can read and write, or `private`, where only the owner can read or write.

The concept of table ownership goes a bit further. The ETS table is intimately linked to the process. If the process dies, the table disappears (and so does all of its content). However, the table can be given away, similar to sockets and their controlling processes, or an heir can be determined so that if the owner process dies, the table is automatically given away to the heir process.

ETS Phone Home

Now that we've covered some ETS concepts, we're ready to move on to the basics of handling tables. ETS functions allow you to create and destroy tables, as well as insert and look up data.

Creating and Deleting Tables

To start an ETS table, call the function `ets:new/2`. This function takes the argument `Name` (as an atom) and then a list of options. In return, you get a unique identifier necessary to use the table, comparable to a pid for processes.

The options can be any of the following:

`Type = set | ordered_set | bag | duplicate_bag`
: This sets the type of table you want to have, as described earlier. The default value is `set`.

`Access = private | protected | public`
: This sets permissions on the table, as described earlier. The default option is `protected`.

named_table

Funnily enough, if you call `ets:new(some_name, [])`, you'll be starting a protected `set` table without a name. For the name to be used as a way to contact a table (and to be made unique), the option `named_table` must be passed to the function. Otherwise, the name of the table will be purely for documentation purposes and will appear in functions such as `ets:i()`, which prints information about all ETS tables in the system.

{keypos, Position}

As you may (and should) recall, ETS tables work by storing tuples. The `Position` parameter holds an integer from 1 to *N*, telling which of each tuple's elements should act as the primary key of the database table. The default key position is set to 1. This means you must be careful if you're using records, as each record's first element will always be the record's name (remember what they look like in their tuple form). If you want to use any field as the key, use `{keypos, #RecordName.FieldName}`, as it will return the position of `FieldName` within the record's tuple representation.

{heir, Pid, Data} | {heir, none}

ETS tables have a process that acts as their parent. If the process dies, the table disappears. If the data attached to a table is something you might want to keep alive, then defining an heir can be useful. If the process attached to a table dies, the heir receives a message saying `{'ETS-TRANSFER', TableId, FromPid, Data}`, where `Data` is the element passed when the option was first defined. The table is automatically inherited by the heir. By default, no heir is defined. It is possible to define or change an heir at a later time by calling `ets:setopts(Table, {heir, Pid, Data})` or `ets:setopts(Table, {heir, none})`. If you simply want to give the table away, call `ets:give_away(Tab, Pid, Data)`.

{read_concurrency, true | false}

This is an option to optimize the table for read concurrency. Setting this option to `true` means that reads become way cheaper to do, but switching to writes becomes a lot more expensive. Basically, this option should be enabled when you do a lot of reading and little writing, and need an extra kick of performance. If you do some reading and some writing, and they are interleaved, using this option might even hurt performance.

{write_concurrency, true | false}

Usually, writing to a table will lock the whole thing and no one else can access it, either for reading or writing, until the writing is done. Setting this option to `true` lets both reads and writes be done concurrently, without affecting the ACID properties of ETS. Doing this, however, will reduce the performance of sequential writes by a single process and also the capacity of concurrent reads. You can combine this option with `read_concurrency` when both writes and reads come in large bursts.

NOTE *ACID stands for* atomicity, consistency, isolation, *and* durability. *ACID properties are those defined for a reliable database transaction system. See* http://en.wikipedia.org/wiki/ACID *for more information.*

`compressed`
: Using this option will allow the data in the table to be compressed for most fields, but not the primary key. This comes at the cost of performance when inspecting entire elements of the table.

Then, the opposite of table creation is table destruction. For that one, all that's needed is to call `ets:delete(Table)`, where `Table` is either a table ID or the name of a named table. If you want to delete a single entry from the table, a very similar function call is required: `ets:delete(Table, Key)`.

Inserting and Looking Up Data

Two other functions are required for basic table handling: `ets:insert(Table, ObjectOrObjects)` and `ets:lookup(Table, Key)`. In the case of `insert/2`, `ObjectOrObjects` can be either a single tuple or a list of tuples to insert.

```
1> ets:new(ingredients, [set, named_table]).
ingredients
2> ets:insert(ingredients, {bacon, great}).
true
3> ets:lookup(ingredients, bacon).
[{bacon,great}]
4> ets:insert(ingredients, [{bacon, awesome}, {cabbage, alright}]).
true
5> ets:lookup(ingredients, bacon).
[{bacon,awesome}]
6> ets:lookup(ingredients, cabbage).
[{cabbage,alright}]
7> ets:delete(ingredients, cabbage).
true
8> ets:lookup(ingredients, cabbage).
[]
```

You'll notice that the `lookup` function returns a list. It will do that for all types of tables, even though set-based tables will always return one item at most. It just means that you should be able to use the `lookup` function in a generic way, even when you use `bag` or `duplicate_bag` tables (which may return many values for a single key).

Another thing that takes place in the preceding snippet is that inserting the same key twice overwrites it. This will always happen in `set` and `ordered_set` tables, but not in `bag` or `duplicate_bag` tables. If you want to avoid this, the function `ets:insert_new/2` might be what you need, as it will insert elements only if they are not already in the table.

NOTE *The tuples do not need to all be the same size in an ETS table, though that is considered the best practice. It is, however, necessary that the tuple is at least of the same size (or greater) than whatever the key position is.*

There's another lookup function available if you need to fetch only part of a tuple: `ets:lookup_element(TableID, Key, PositionToReturn)`. It will return the element that matched (or a list of them if there is more than one in a `bag` or `duplicate_bag` table). If the element isn't there, the function errors out, with `badarg` as a reason.

Let's try our previous example again with a `bag` table:

```
9> TabId = ets:new(ingredients, [bag]).
16401
10> ets:insert(TabId, {bacon, delicious}).
true
11> ets:insert(TabId, {bacon, fat}).
true
12> ets:insert(TabId, {bacon, fat}).
true
13> ets:lookup(TabId, bacon).
[{bacon,delicious},{bacon,fat}]
```

As this is a `bag` table, `{bacon, fat}` is there only once, even though we inserted twice, but you can see that we can still have more than one `bacon` entry. The other thing to look at here is that without passing in the `named_table` option, we must use `TabId` to use the table.

NOTE *If at any point while copying these examples your shell crashes, the tables are going to disappear, as their parent process (the shell) has disappeared.*

The last basic operations we can make use of will be about traversing table entries one by one, `ordered_set` tables are the best fit for this task:

```
14> ets:new(ingredients, [ordered_set, named_table]).
ingredients
15> ets:insert(ingredients,
15>            [{ketchup, "not much"}, {mustard, "a lot"},
15>             {cheese, "yes", "goat"}, {patty, "moose"},
15>             {onions, "a lot", "caramelized"}]).
true
16> Res1 = ets:first(ingredients).
cheese
17> Res2 = ets:next(ingredients, Res1).
ketchup
18> Res3 = ets:next(ingredients, Res2).
mustard
19> ets:last(ingredients).
patty
20> ets:prev(ingredients, ets:last(ingredients)).
onions
```

As you can see, elements are now in sorting order, and they can be accessed one after the other, both forward and backward.

Oh yeah, we need to see what happens in boundary conditions:

```
21> ets:next(ingredients, ets:last(ingredients)).
'$end_of_table'
22> ets:prev(ingredients, ets:first(ingredients)).
'$end_of_table'
```

When you see atoms starting with a `$`, you should know that they're some special value (chosen by convention by the OTP team) telling you about something. Whenever you're trying to iterate outside the table, you'll see these `$end_of_table` atoms. Other than cases like this, you should avoid using atoms starting with `$` in order to prevent confusion or possible clashes with OTP's own atoms.

We've now covered how to use ETS as a basic key/value store. Next, we'll look at some more advanced uses.

Meeting Your Match

What if you want to do more than just match on keys when doing lookups? When you think about it, the best way to select things would be with pattern matching, right? The ideal scenario would be to somehow store a pattern to match on within a variable (or as a data structure), pass that to some ETS function, and let said function do its thing.

This is called *higher-order pattern matching*, and sadly, it is not available in Erlang. In fact, very few languages have it. Instead, Erlang has a kind of sublanguage that Erlang programmers have agreed to use to describe pattern matching as a bunch of regular data structures.

This notation is based on tuples to fit nicely with ETS. It simply lets you specify variables (regular and "don't care" variables), that can be mixed with the tuples to do pattern matching. Variables are written as `'$0'`, `'$1'`, `'$2'`, and so on (the number has no importance except in how you'll get the results) for regular variables. The "don't care" variable can be written as `'_'`. All these atoms (`'_'`, `'$0'`, `'$1'`, and so on) can be used to represent the pattern for a table entry when placed in a tuple like so:

```
{items, '$3', '$1', '_', '$3'}
```

This is roughly equivalent to saying `{items, C, A, _, C}` with regular pattern matching. As such, you can guess that the first element needs to be the atom `items`, the second and fifth slots of the tuple need to be identical, and so on.

To make use of this notation in a more practical setting, two functions are available: `match/2` and `match_object/2`. (`match/3` and `match_object/3` are available as well, but their use is outside the scope of this chapter—readers are encouraged to check the docs for details.)

The `match/2` function returns the variables of the pattern, and the `match_object/2` function returns the whole entry that matched the pattern:

```
1> ets:new(table, [named_table, bag]).
table
2> ets:insert(table, [{items, a, b, c, d}, {items, a, b, c, a},
2>                    {cat, brown, soft, loveable, selfish},
2>                    {friends, [jenn,jeff,etc]}, {items, 1, 2, 3, 1}]).
true
3> ets:match(table, {items, '$1', '$2', '_', '$1'}).
[[a,b],[1,2]]
4> ets:match(table, {items, '$114', '$212', '_', '$6'}).
[[d,a,b],[a,a,b],[1,1,2]]
5> ets:match_object(table, {items, '$1', '$2', '_', '$1'}).
[{items,a,b,c,a},{items,1,2,3,1}]
6> ets:delete(table).
true
```

The nice thing about `match/2` as a function is that it returns only what is strictly necessary. This is useful because, as mentioned earlier, ETS tables follow the nothing-shared ideals. If you have very large records, copying only the necessary fields might be a good thing to do.

Anyway, you'll also notice that while the numbers in variables have no explicit meaning, their order is important. In the final list of values returned, the value bound to `$114` will always come after the values bound to `$6` by the pattern. If nothing matches, empty lists are returned.

It is also possible that you might want to delete entries based on such a pattern match. In these cases, the function `ets:match_delete(Table, Pattern)` is what you want.

This is all fine and lets us do pattern matching of any literal value, albeit in a funny-looking way. It would be pretty neat if it were possible to have things like comparisons and ranges, explicit ways to format the output (maybe lists aren't what we want), and so on. Oh wait, you can!

You Have Been Selected

Erlang provides an approach that gives us something more equivalent to true function heads–level pattern matching, including very simple guards. If you've ever used a SQL database, you might have seen ways to do queries where you compare elements that are greater than, equal to, or smaller than other elements. This is the kind of good stuff we want here.

The people behind Erlang thus took the syntax for matches and augmented it in crazy ways until it was powerful enough. Sadly, they also made it unreadable. Here's what it can look like:

```
[{{'$1','$2',<<1>>,'$3','$4'},
  [{'andalso',{'>','$4',150},{'<','$4',500}},
   {'orelse',{'==','$2',meat},{'==','$2',dairy}}],
  ['$1']},
 {{'$1','$2',<<1>>,'$3','$4'},
  [{'<','$3',4.0},{is_float,'$3'}],
  ['$1']}]
```

This is pretty ugly—not the data structure you would want your children to look like. Believe it or not, we'll learn how to write these things called *match specifications*, but not in the preceding form—no, that would be a bit too hard for no reason. But we'll still learn how to read them, though!

Here's what match specifications look like from a higher-level view:

```
[{InitialPattern1, Guards1, ReturnedValue1},
 {InitialPattern2, Guards2, ReturnedValue2}].
```

And from a yet higher view, we see this:

```
[Clause1,
 Clause2]
```

These things represent, roughly, the pattern in a function head, then the guards, and then the body of a function. The format is still limited to `'$N'` variables for the initial pattern, just as it was for match functions. The new sections are the guard patterns, which allow us to do something quite similar to regular guards. If we look at the guard `[{'<','$3',4.0},{is_float,'$3'}]`, we can see that it is quite similar to `... when Var < 4.0, is_float(Var) -> ...` as a guard.

The next guard, more complex this time, is as follows:

```
[{'andalso',{'>','$4',150},{'<','$4',500}},
 {'orelse',{'==','$2',meat},{'==','$2',dairy}}]
```

Translating it gives us a guard that looks like `... when Var4 > 150 andalso Var4 < 500, Var2 == meat orelse Var2 == dairy -> ...`. Got it?

Each operator or guard function works with a prefix syntax, meaning that we use the order `{FunctionOrOperator, Arg1, ..., ArgN}`. So `is_list(X)` becomes `{is_list, '$1'}`, `X andalso Y` becomes `{'andalso', X, Y}`, and so on. Reserved keywords such as `andalso` and `orelse`, and operators like `==`, need to be turned into atoms so the Erlang parser won't choke on them.

The last section of the pattern is what you want to return. Just put the variables you need in there. If you want to return the full input of

the match specification, use the variable '$_' to do so. A full specification of match specifications can be found in the Erlang documentation at *http://www.erlang.org/doc/apps/erts/match_spec.html*.

As I said earlier, we won't learn how to write patterns that way, since there's a nicer way to do it. ETS comes with what is called a *parse transform*. A parse transform is an underdocumented (and officially not supported by the OTP team for whatever problems you encounter) way of accessing the Erlang parse tree halfway through the compiling phase. They let ballsy Erlang programmers transform the code in a module to a new alternative form. A parse transform can be pretty much anything and change existing Erlang code to almost anything else, as long as it doesn't change the language's syntax or tokens.

The parse transform coming with ETS must be enabled manually for each module that needs it. The way to do it in a module is as follows:

```
-module(SomeModule).
-include_lib("stdlib/include/ms_transform.hrl").
...
some_function() ->
    ets:fun2ms(fun(X) when X > 4 -> X end).
```

The line `-include_lib("stdlib/include/ms_transform.hrl").` contains some special code that will override the meaning of `ets:fun2ms(SomeLiteralFun)` whenever it's being used in a module. Rather than being a higher-order function, the parse transform will analyze what is in the `fun` (the pattern, the guards, and the return value), remove the function call to `ets:fun2ms/1`, and replace it all with an actual match specification. Weird, huh? The best thing is that because this happens at compile time, there is no overhead to using this way of doing things.

We can try it in the shell, without the include file this time:

```
1> ets:fun2ms(fun(X) -> X end).
[{'$1',[],['$1']}]
2> ets:fun2ms(fun({X,Y}) -> X+Y end).
[{{'$1','$2'},[],[{'+','$1','$2'}]}]
3> ets:fun2ms(fun({X,Y}) when X < Y -> X+Y end).
[{{'$1','$2'},[{'<','$1','$2'}],[{'+','$1','$2'}]}]
4> ets:fun2ms(fun({X,Y}) when X < Y, X rem 2 == 0 -> X+Y end).
[{{'$1','$2'},
  [{'<','$1','$2'},{'==',{'rem','$1',2},0}],
  [{'+','$1','$2'}]}]
5> ets:fun2ms(fun({X,Y}) when X < Y, X rem 2 == 0; Y == 0 -> X end).
[{{'$1','$2'},
  [{'<','$1','$2'},{'==',{'rem','$1',2},0}],
  ['$1']},
 {{'$1','$2'},[{'==','$2',0}],['$1']}]
```

All of these are written so easily now! And, of course, the `funs` are much simpler to read.

How about that complex example from the beginning of the section? Here's what it would be like as a `fun`:

```
6> ets:fun2ms(fun({Food, Type, <<1>>, Price, Calories})
6>   when Calories > 150 andalso Calories < 500,
6>        Type == meat orelse Type == dairy;
6>        Price < 4.00, is_float(Price) ->
6> Food end).
[{{'$1','$2',<<1>>,'$3','$4'},
  [{'andalso',{'>','$4',150},{'<','$4',500}},
   {'orelse',{'==','$2',meat},{'==','$2',dairy}}],
  ['$1']},
 {{'$1','$2',<<1>>,'$3','$4'},
  [{'<','$3',4.0},{is_float,'$3'}],
  ['$1']}]
```

The match specification doesn't entirely make sense at first glance, but at least it's much simpler to figure out what it means when the variables actually have names rather than numbers.

One thing to be careful about is that not all `fun`s are valid match specifications:

```
7> ets:fun2ms(fun(X) -> my_own_function(X) end).
Error: fun containing the local function call 'my_own_function/1' (called in body)
 cannot be translated into match_spec
{error,transform_error}
8> ets:fun2ms(fun(X,Y) -> ok end).
Error: ets:fun2ms requires fun with single variable or tuple parameter
{error,transform_error}
9> ets:fun2ms(fun([X,Y]) -> ok end).
Error: ets:fun2ms requires fun with single variable or tuple parameter
{error,transform_error}
10> ets:fun2ms(fun({<<X/binary>>}) -> ok end).
Error: fun head contains bit syntax matching of variable 'X', which cannot be
 translated into match_spec
{error,transform_error}
```

The function head needs to match on a single variable or a tuple, no nonguard functions can be called as part of the return value, and assigning values from within binaries is not allowed. Try some stuff in the shell to see what you can do.

To make match specifications useful, it would make sense to use them. This can be done with these functions:

- `ets:select/2` to fetch results
- `ets:select_reverse/2` to get results in reverse in `ordered_set` tables (for other types, it's the same as `select/2`)
- `ets:select_count/2` to know how many results match the specification
- `ets:select_delete(Table, MatchSpec)` to delete records matching a match specification

DON'T DRINK TOO MUCH KOOL-AID

A function like `ets:fun2ms` sounds totally awesome, right? But you need to be careful with it! A problem with it is that if `ets:fun2ms` can handle dynamic `funs` when in the shell (you can pass `funs` around, and it will just eat them up), this isn't possible in compiled modules. This is due to the fact that Erlang has two kinds of `funs`: shell `funs` and module `funs`.

Module `funs` are compiled down to some compact format understood by the VM. They're opaque and cannot be inspected to see how they are on the inside.

On the other hand, shell `funs` are abstract terms not yet evaluated. They're made in a way that allows the shell to call the evaluator on them. The function `fun2ms` will thus have two versions of itself: one for when you're getting compiled code and one for when you're in the shell.

This is fine, except that the `funs` aren't interchangeable with different types of `funs`. This means that you can't take a compiled `fun` and try to call `ets:fun2ms` on it while in the shell, and you can't take a dynamic `fun` and send it over to a compiled bit of code that's calling `fun2ms` in there. Too bad!

Let's try it. We'll define a record for our tables, and then populate them with various goods:

```
11> rd(food, {name, calories, price, group}).
food
12> ets:new(food, [ordered_set, {keypos,#food.name}, named_table]).
food
13> ets:insert(food, [#food{name=salmon, calories=88, price=4.00, group=meat},
13>  #food{name=cereals, calories=178, price=2.79, group=bread},
13>  #food{name=milk, calories=150, price=3.23, group=dairy},
13>  #food{name=cake, calories=650, price=7.21, group=delicious},
13>  #food{name=bacon, calories=800, price=6.32, group=meat},
13>  #food{name=sandwich, calories=550, price=5.78, group=whatever}]).
true
```

We can then try to select food items under a given number of calories:

```
14> ets:select(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 -> N end)).
[#food{name = cereals,calories = 178,price = 2.79,group = bread},
 #food{name = milk,calories = 150,price = 3.23,group = dairy},
 #food{name = salmon,calories = 88,price = 4.0,group = meat},
 #food{name = sandwich,calories = 550,price = 5.78,group = whatever}]
15> ets:select_reverse(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 -> N end)).
[#food{name = sandwich,calories = 550,price = 5.78,group = whatever},
 #food{name = salmon,calories = 88,price = 4.0,group = meat},
 #food{name = milk,calories = 150,price = 3.23,group = dairy},
 #food{name = cereals,calories = 178,price = 2.79,group = bread}]
```

Or maybe what we want is just delicious food:

```
16> ets:select(food, ets:fun2ms(fun(N = #food{group=delicious}) -> N end)).
[#food{name = cake,calories = 650,price = 7.21,group = delicious}]
```

Deleting has a little special twist to it. You need to return `true` in the pattern instead of any kind of value:

```
17> ets:select_delete(food, ets:fun2ms(fun(#food{price=P}) when P > 5 -> true end)).
3
18> ets:select_reverse(food, ets:fun2ms(fun(N = #food{calories=C}) when C < 600 -> N end)).
[#food{name = salmon,calories = 88,price = 4.0,group = meat},
 #food{name = milk,calories = 150,price = 3.23,group = dairy},
 #food{name = cereals,calories = 178,price = 2.79,group = bread}]
```

As the last selection shows, items over $5.00 were removed from the table.

ETS has a lot more functions, such as ways to convert the table to lists or files (`ets:tab2list/1`, `ets:tab2file/1`, `ets:file2tab/1`) and get information about all tables (`ets:i/0`, `ets:info(Table)`). Head over to the official documentation to learn more about these functions.

There's also a module called `tv` (Table Viewer) that can be used to visually manage the ETS tables on a given Erlang VM. Just call `tv:start()`, and a window will be opened, showing your tables.

DETS

DETS is a disk-based version of ETS, with a few key differences:

- There are no longer `ordered_set` tables.
- There is a disk-size limit of 2GB for DETS files.
- Operations such as `prev/1` and `next/1` are not nearly as safe or fast.
- Starting and stopping tables has changed a bit. A new database table is created by calling `dets:open_file/2`, and is closed by calling `dets:close/1`. The table can later be reopened by calling `dets:open_file/1`.

Otherwise, the API is nearly the same, and it is thus possible to have a simple way to handle writing and looking for data inside files.

DON'T DRINK TOO MUCH KOOL-AID

DETS risks being slow, as it is a disk-only database. You might feel like coupling ETS and DETS tables into a somewhat efficient database that stores both in RAM and on disk.

If you feel like doing so, it might be a good idea to look into Mnesia (covered in Chapter 29) as a database, which does exactly the same thing, while adding support for sharding, transactions, and distribution.

A Little Less Conversation, a Little More Action, Please

Following this rather long section title (and long previous sections), we'll turn to the practical problem that brought us here in the first place: updating `regis` so that it uses ETS and gets rid of a few potential bottlenecks.

Before we get started, we need to think about how we're going to handle operations, and what is safe and unsafe. Things that should be safe are those that modify nothing and are limited to one query (not three or four over time). They can be done by anyone at any time. Everything else that has to do with writing to a table, updating records, deleting them, or reading in a way that requires consistency over many requests is to be considered unsafe.

Because ETS has no transactions whatsoever, all unsafe operations should be performed by the process that owns the table. The safe ones should be allowed to be public—done outside the owner process. We'll keep this in mind as we update `regis`.

The first step will be to make a copy of *regis-1.0.0* as *regis-1.1.0* (you can get a copy of *regis-1.1.0* at *http://learnyousomeerlang.com/static/erlang/regis-1.1.0.zip*). I'm bumping up the second number and not the third one here because our changes shouldn't break the existing interface and are technically not bug fixes, so we're going to consider this version to be only a feature upgrade.

The Interface

In that new directory, we'll need to operate only on *regis_server.erl* at first. We'll keep the interface intact so all the rest, in terms of structure, should not need to change too much.

```
%%% The core of the app: the server in charge of tracking processes.
-module(regis_server).
-behavior(gen_server).
-include_lib("stdlib/include/ms_transform.hrl").

-export([start_link/0, stop/0, register/2, unregister/1, whereis/1,
         get_names/0]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         code_change/3, terminate/2]).

%%%%%%%%%%%%%%%%%
%%% INTERFACE %%%
%%%%%%%%%%%%%%%%%
start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).
```

```
stop() ->
    gen_server:call(?MODULE, stop).

%% Give a name to a process.
register(Name, Pid) when is_pid(Pid) ->
    gen_server:call(?MODULE, {register, Name, Pid}).

%% Remove the name from a process.
unregister(Name) ->
    gen_server:call(?MODULE, {unregister, Name}).

%% Find the pid associated with a process.
whereis(Name) -> ok.

%% Find all the names currently registered.
get_names() -> ok.
```

For the public interface, only `whereis/1` and `get_names/0` will change and be rewritten. That's because, as mentioned earlier, they are single-read safe operations. The rest will require to be serialized in the process owning the table.

That's it for the API so far. Let's head for the inside of the module.

Implementation Details

We're going to use an ETS table to store stuff, so it makes sense to put that table into the `init` function. Moreover, because our `whereis/1` and `get_names/0` functions will give public access to the table (for speed reasons), naming the table will be necessary for it to be accessible to the outside world. By naming the table, much like what happens when we name processes, we can hardcode the name in the functions, so we won't need to pass around an ID.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% GEN_SERVER CALLBACKS %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
init([]) ->
    ?MODULE = ets:new(regis, [set, named_table, protected]),
    {ok, ?MODULE}.
```

The next function will be `handle_call/3`, handling the message `{register, Name, Pid}` as defined in `register/2`.

```
handle_call({register, Name, Pid}, _From, Tid) ->
    %% Neither the name or the pid can already be in the table
    %% so we match for both of them in a table-long scan using this.
    MatchSpec = ets:fun2ms(fun({N,P,_Ref}) when N==Name; P==Pid -> {N,P} end),
    case ets:select(Tid, MatchSpec) of
        [] -> % free to insert
            Ref = erlang:monitor(process, Pid),
            ets:insert(Tid, {Name, Pid, Ref}),
            {reply, ok, Tid};
```

```
        [{Name,_}|_] -> % maybe more than one result, but name matches
            {reply, {error, name_taken}, Tid};
        [{_,Pid}|_] -> % maybe more than one result, but Pid matches
            {reply, {error, already_named}, Tid}
    end;
```

This is by far the most complex function in the module. There are three basic rules to respect:

- A process cannot be registered twice.
- A name cannot be taken twice.
- A process can be registered if it doesn't break the first two rules.

This is what the preceding code does. The match specification derived from `fun({N,P,_Ref}) when N==Name; P==Pid -> {N,P} end` will look through the whole table for entries that match either the name or the pid that we're trying to register. If there's a match, we return both the name and pid that were found. This may be weird, but it makes sense to want both when we look at the patterns for the `case ... of` after that.

The first pattern means nothing was found, and so insertions are good. We monitor the process we have registered (to unregister it in case of failure), and then add the entry to the table. In case the name we are trying to register was already in the table, the pattern `[{Name,_}|_]` will take care of it. If it was the pid that matched, then the pattern `[{_,Pid}|_]` will take care of it. That's why both values are returned: It's simpler to match on the whole tuple later on, not caring whether it was `Pid` or `Name` that matched in the match specifications.

Why is the pattern of the form `[Tuple|_]` rather than just `[Tuple]`? The explanation is simple enough: If we're traversing the table looking for either pids or names that are similar, it is possible the list returned will be `[{NameYouWant, SomePid},{SomeName,PidYouWant}]`. If that happens, then a pattern match of the form `[Tuple]` will crash the process in charge of the table and ruin your day.

Oh yeah, don't forget to add the `-include_lib("stdlib/include/ms_transform.hrl").` in the module; otherwise, `fun2ms` will die with a weird error message:

```
** {badarg,{ets,fun2ms,
            [function,called,with,real,'fun',should,be,transformed,with,
             parse_transform,'or',called,with,a,'fun',generated,in,the,
             shell]}}
```

That's what happens when you forget the include file. Consider yourself warned. Look before crossing the streets, don't cross the streams, and don't forget your include files.

The next bit handles when we ask to manually unregister a process:

```
handle_call({unregister, Name}, _From, Tid) ->
    case ets:lookup(Tid, Name) of
        [{Name,_Pid,Ref}] ->
            erlang:demonitor(Ref, [flush]),
            ets:delete(Tid, Name),
            {reply, ok, Tid};
        [] ->
            {reply, ok, Tid}
    end;
```

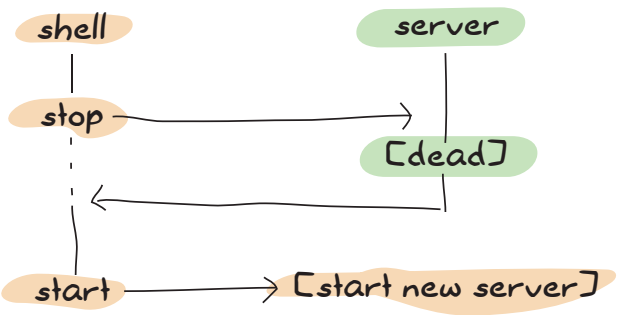
This is similar to what's in the old version of the code. The idea is simple: Find the monitor reference (with a lookup on the name), cancel the monitor, and then delete the entry and keep going. If the entry is not there, we pretend we deleted it anyway, and everyone will be happy. Oh, how dishonest we are.

The next bit is about stopping the server:

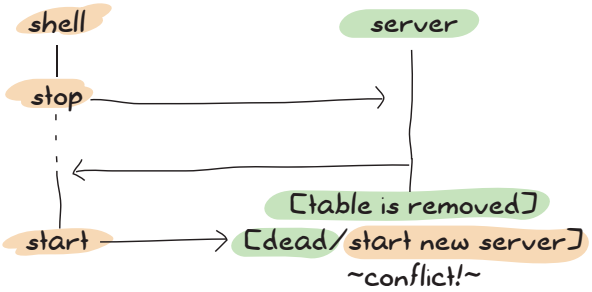
```
handle_call(stop, _From, Tid) ->
    %% For the sake of being synchronous and because emptying ETS
    %% tables might take a bit longer than dropping data structures
    %% held in memory, dropping the table here will be safer for
    %% tricky race conditions, especially in tests where we start/stop
    %% servers a lot. In regular code, this doesn't matter.
    ets:delete(Tid),
    {stop, normal, ok, Tid};
handle_call(_Event, _From, State) ->
    {noreply, State}.
```

As the comments in the code say, we could have been fine just ignoring the table and letting it be garbage-collected. However, because the test suite we wrote in Chapter 24 starts and stops the server all the time, delays can be a bit dangerous.

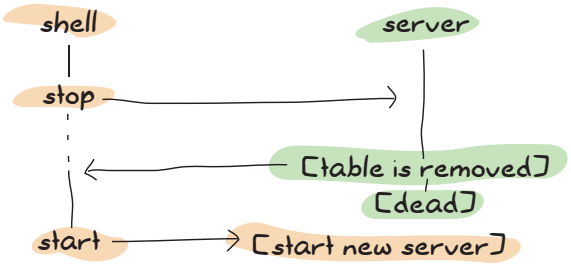
This is what the timeline of the process looks like with the old one:



And here's what sometimes happens with the new one:



By using this scheme, we're making it a lot more unlikely for errors to happen by doing more work in the synchronous part of the code:



If you don't plan on running the test suite very often, you can just ignore the whole thing. I've decided to show it to avoid nasty surprises, although in a non-test system, this kind of edge case should rarely occur.

Here are the other OTP callbacks:

```
handle_cast(_Event, State) ->
    {noreply, State}.

handle_info({'DOWN', Ref, process, _Pid, _Reason}, Tid) ->
    ets:match_delete(Tid, {'_', '_', Ref}),
    {noreply, Tid};
handle_info(_Event, State) ->
    {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.
```

We don't care about any of them, except receiving a `DOWN` message, meaning one of the processes we were monitoring died. When that happens, we delete the entry based on the reference we have in the message, and then move on.

You'll notice that `code_change/3` could actually work as a transition between the old `regis_server` and the new `regis_server`. Implementing this function is left as an exercise for the reader. I hate books that give exercises for the reader without solutions, so here's at least a little pointer so I'm not just being a jerk like all the other writers out there: You need to take either of the two GB trees from the older version, and use `gb_trees:map/2` or the `gb_trees` iterators to populate a new table before moving on. The downgrade function can be written by doing the opposite.

All that's left to do is fix the two public functions we have left unimplemented. Of course, we could write a `%% TODO` comment, call it a day, and go drink until we forget we're programmers, but that would be a bit irresponsible. Let's fix stuff:

```
%% Find the pid associated with a process.
whereis(Name) ->
    case ets:lookup(?MODULE, Name) of
        [{Name, Pid, _Ref}] -> Pid;
        [] -> undefined
    end.
```

This one looks for a name, and returns the pid or `undefined` depending on whether the entry has been found or not. Note that we use `regis` (`?MODULE`) as the table name here—that's why we made it protected and named in the first place.

Here's the next one:

```
%% Find all the names currently registered.
get_names() ->
    MatchSpec = ets:fun2ms(fun({Name, _, _}) -> Name end),
    ets:select(?MODULE, MatchSpec).
```

We use `fun2ms` again to match on the name and keep only that. Selecting from the table will return a list and do what we need.

That's it! You can run the test suite in `test/` to make things go:

```
$ erl -make
... <snip> ...
Recompile: src/regis_server
$ erl -pa ebin
... <snip> ...
1> eunit:test(regis_server).
  All 13 tests passed.
ok
```

Hell, yes—I think we can consider ourselves pretty good at ETSing now.

You know what would be really nice to do next? We could actually explore the distributed aspects of Erlang. Maybe we can bend our minds in a few more twisted ways before being finished with the Erlang beast. Let's see.

26

DISTRIBUNOMICON

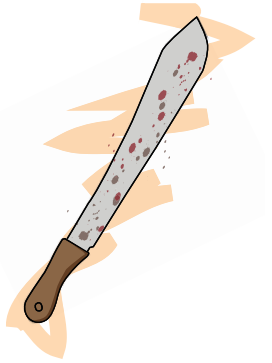
Oh, hi! Please have a seat. I was expecting you.

When you first heard of Erlang, there were two or three attributes that likely attracted you. Erlang is a functional language, it has great semantics for concurrency, and it supports distribution. We've covered the first two attributes, and spent time exploring a dozen more you possibly didn't expect. Now we're at the last big thing: distribution.

We've waited quite a while before getting here because it's not exactly useful to get distributed if we can't make things work locally to begin with. We're finally up to the task and have come a long way to get where we are.

Like almost every other feature of Erlang, the distributed layer of the language was first added in order to provide fault tolerance. Software running on a single machine is always at risk of having that single machine dying

and taking your application offline. Software running on many machines makes it easier to handle hardware failure if, and only if, the application was built correctly. There is no benefit regarding fault tolerance if your application runs on many servers but cannot deal with one of them being taken down.



Distributed programming is like being left alone in the dark, with monsters everywhere. It's scary, and you don't know what to do or what's coming at you. Bad news: Distributed Erlang is still leaving you alone in the dark to fight the scary monsters. It won't do any of that kind of hard work for you. Good news: Instead of being alone with nothing but pocket change and a poor sense of aim (making it harder to kill the monsters), Erlang gives you a flashlight, a machete, and a pretty kick-ass mustache to feel more confident (this also applies to female readers).

That's not especially due to how Erlang is written, but more or less due to the nature of distributed software. Erlang provides the few basic building blocks of distribution: ways to have many nodes (VMs) communicating with each other, serializing and deserializing data in communications, extending the concepts of multiple processes to many nodes, ways to monitor network failures, and so on. However, it will not provide solutions to software-specific problems such as "what happens when stuff crashes."

This is the standard "tools, not solutions" approach you've seen before in OTP. You rarely get full-blown software and applications, but you do get many components you can use to build systems. You'll have tools that tell you when parts of the system go up or down and tools to do a bunch of stuff over the network, but hardly any silver bullet that takes care of fixing things for you.

Let's see what kind of flexing we can do with these tools.

This Is My Boomstick

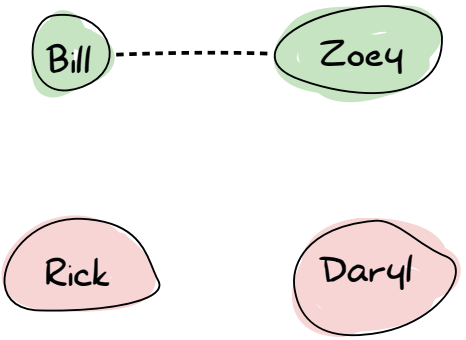
To tackle all these monsters in the dark, we've been granted a very useful thing: pretty complete network transparency.

An instance of an Erlang VM that is up and running, ready to connect to other VMs, is called a *node*. Whereas some languages or communities will consider a server to be a node, in Erlang, each VM is a node. You can have 50 nodes running on a single computer, or 50 nodes running on 50 computers. It doesn't really matter.

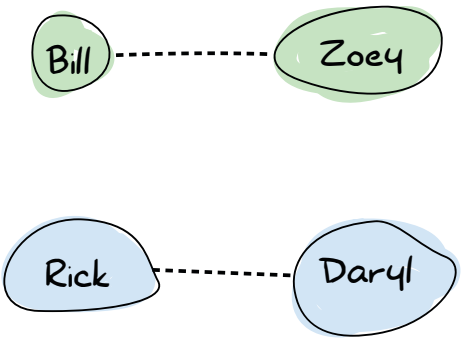
When you start a node, you give it a name, and it will connect to an application called Erlang Port Mapper Daemon (EPMD), which will run on each of the computers that are part of your Erlang cluster. EPMD will act as a name server that lets nodes register themselves, contact other nodes by name rather than port numbers, and warn you about any name clashes.

From this point on, a node can decide to set up a connection to another node. When it does so, both nodes automatically start monitoring each other, and they can tell if the connection is dropped or a node disappears. More importantly, when a new node joins another node that is already part of a group of nodes connected together, the new node automatically connects to the entire group.

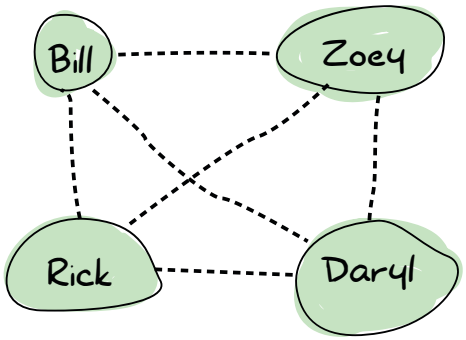
To illustrate how Erlang nodes set up their connections, let's take the idea of a group of survivors during a zombie outbreak. The survivors are Zoey, Bill, Rick, and Daryl. Zoey and Bill know each other and communicate on the same frequency on walkie-talkies. Rick and Daryl are each on their own, as shown here:



Now let's say Rick and Daryl meet on their way to a survivor camp. They share their walkie-talkie frequency and can now stay up to date with each other before splitting ways again, like this:



At some point, Rick meets Bill. Both are pretty happy about that, and so they decide to share frequencies. At this point, the connections spread, and the final graph looks like this:



That means that any survivor can contact any other directly. This is useful because in the event of the death of any survivor, no one is left isolated. Erlang nodes are set up in this way: Everyone connects to everyone.

DON'T DRINK TOO MUCH KOOL-AID

Connecting nodes in this manner, while nice for some fault-tolerance reasons, has a pretty bad drawback with regard to how much you can scale. It will be hard to have hundreds and hundreds of nodes part of your Erlang cluster, simply because of how many connections are required and how much chatter is involved. In fact, you will require one ephemeral port per node. While some large-scale projects have managed to do this, you may find it easier to just split groups of nodes into smaller clusters.

If you were planning on using Erlang to do that kind of heavy setup, please read on in this chapter to see why things are that way and what might be done to work around the problem, if possible.

Once the nodes are connected, they remain fully independent. They keep their own process registry and ETS tables (with their own names for tables), and the modules they load are independent from each other. A connected node that crashes won't bring down the other nodes in the cluster.

Connected nodes can then start exchanging messages. The distribution model of Erlang was designed so that local processes can contact remote processes and send them regular messages. How is this possible if nothing is shared and all the process registries are unique? As we'll see when we get into the specifics of distribution in "Connecting Nodes" on page 459, there is a way to access registered processes on a particular node. From that point on, a first message can be sent.

Erlang messages are serialized and unserialized automatically for you in a transparent manner. All data structures, including pids, will work the

same remotely and locally. This means that we can send pids over the network, and then the other side can communicate with them by sending messages. Even better, links and monitors can be set up across the network if you can access the pids!

So if Erlang is doing so much to make everything that transparent, why am I saying it's only giving us a machete, a flashlight, and a mustache?

Fallacies of Distributed Computing

Much like a machete is meant to kill only a given type of monster, Erlang's tools are meant to handle only some kinds of distributed computing. To understand the tools Erlang gives us, it will be useful to first have an idea of what kind of landscape exists in the distributed world, and which assumptions Erlang makes in order to provide fault tolerance.

Some very smart guys took their time in the past few decades to categorize the kind of stuff that goes wrong with distributed computing. They came up with eight major assumptions people make (some of which Erlang's designers made for various reasons) that end up biting them in the ass later.

NOTE *The fallacies of distributed computing were introduced in "Fallacies of Distributed Computing Explained," by Arnon Rotem-Gal-Oz, available online at* http://www.rgoarchitects.com/Files/fallacies.pdf.

The Network Is Reliable

The first fallacy of distributed computing is assuming that the application can be distributed over the network. That's kind of weird to say, but there will be plenty of times where the network will go down for annoying reasons: power failures, broken hardware, someone tripping over a cord, vortex to other dimensions engulfing mission-critical components, headcrabs infestation, copper theft, and so on.

One of the biggest errors you can make is to think you can always reach remote nodes and talk to them. This is somewhat possible to handle by adding more hardware and gaining redundancy, so that if hardware fails, the application can still be reached somewhere else. The other thing to do is to be ready to suffer a loss of messages and requests, and also be ready for things to become unresponsive. This is especially true when you depend on some kind of third-party service that's no longer there, while your own software stack keeps working well.

Erlang doesn't have any special measures to deal with network reliability, as usually the decisions made will be application specific. After all, who else but you can know how important a specific component will be? Still, you're not totally alone, as a distributed Erlang node will be able to detect other nodes getting disconnected (or becoming unresponsive). There are specific functions to monitor nodes, and links and monitors will also be triggered upon a disconnection.

The best thing Erlang has in this case is its asynchronous communication mode. By sending messages asynchronously and forcing developers to send a reply back when things work well, Erlang pushes for all message-passing activities to intuitively handle failure. If the process you're talking to is on a node that disappears due to some network failure, you handle it as naturally as any local crash. This is one of the many reasons why Erlang is said to scale well (scaling not only in performance but also in design).

DON'T DRINK TOO MUCH KOOL-AID

Linking and monitoring across nodes can be dangerous. In the case of a network failure, all remote links and monitors are triggered at once. This might then generate thousands and thousands of signals and messages to various processes, which puts a heavy and unexpected load on the system.

Preparing for an unreliable network also means preparing for sudden failures and making sure your system doesn't get crippled by part of the system suddenly disappearing.

There Is No Latency

One of the double-edged aspects of seemingly good distribution systems is that they often end up hiding the fact that the function calls you are making are remote. While you expect some function calls to be really fast, that will not be the case when they are made over the network. It's the difference between ordering a pizza from within the pizzeria and getting one delivered from another city to your house. While there will always be a basic wait time, it's possible that your pizza will be delivered cold because delivery took too long.

Forgetting that network communications make things slower even for really small messages can be a costly error if you always expect really fast results. Erlang's model treats us well there. Because of the way we set up our local applications with isolated processes, asynchronous messages, and timeouts, and we're always thinking about the possibility of processes failing, there is little adaptation required to go distributed. The timeouts, links, monitors, and asynchronous patterns remain the same, and still are as reliable. Erlang doesn't implicitly assume there is no latency, and in fact, always expects it by design.

You, however, might make that assumption in your design and expect replies faster than realistically possible. Just keep an eye open.

Bandwidth Is Infinite

Although network transfers are getting faster and faster all the time, and generally speaking, each byte transferred over the network is cheaper as time goes by, it is risky to assume that sending copious amounts of data is simple and easy.

Because of how we build applications locally, we generally won't have too many problems with that in Erlang. Remember that one good trick is to send messages about what is happening, rather than moving new states around ("player X found item Y," rather than sending player X's entire inventory over and over again).

If, for some reason, you need to send large messages, be extremely careful. The way Erlang distribution and communication work over many nodes is especially sensitive to large messages. If two nodes are connected, all of their communications will tend to happen over a single TCP connection. Because we generally want to maintain message ordering between two processes (even across the network), messages will be sent sequentially over the connection. That means that if you have one very large message, you might be blocking the channel for all the other messages.

Worse than that, Erlang knows whether nodes are alive by sending *heartbeats*. Heartbeats are small messages sent at a regular interval between two nodes, basically saying, "I'm still alive. Keep on keepin' on!" They're like our zombie survivors routinely pinging each other with messages. "Bill, are you there?" And if Bill never replies, you might assume he is dead (or out of batteries), and he won't get your future communications. Heartbeats are sent over the same channel as regular messages.

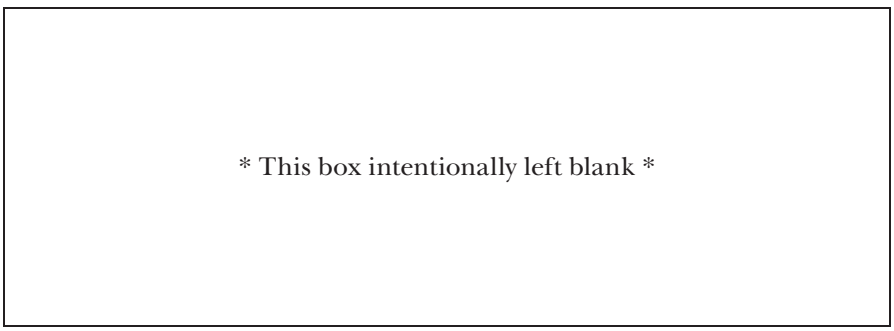
The problem is that a large message can hold back heartbeats. Too many large messages keeping heartbeats at bay for too long will eventually result in either of the nodes assuming the other is unresponsive and closing its connection to it. That's bad.

The good Erlang design lesson to keep in mind to prevent such problems is to keep your messages small. Everything will be better that way.

The Network Is Secure

When you get distributed, it's often very dangerous to believe that everything is safe—that you can trust the messages you receive. It can be simple things like someone fabricating messages and sending them to you, someone intercepting packets and modifying them (or looking at sensitive data), or in the worst case, someone being able to take over your application or the system on which it runs.

In the case of distributed Erlang, this is sadly an assumption that was made. Here is what Erlang's security model looks like:



Yep. This is because Erlang distribution was initially meant for fault tolerance and redundancy of components. In the old days of the language, back when it was used for telephone switches and other telecommunication applications, Erlang would often be deployed on hardware running in the weirdest places—very remote locations with odd conditions (engineers sometimes needed to attach servers to the wall to avoid wet ground or install custom heating systems in the woods in order for the hardware to run at optimal temperatures). In these cases, you had failover hardware in the same physical location as the main hardware. This is often where distributed Erlang would run, and it explains why Erlang designers assumed a safe network to operate with.

Sadly, this means that modern Erlang applications can rarely be clustered automatically across different data centers. In fact, it isn't recommended to do so. Most of the time, you will want your system to be based on many smaller, walled-off clusters of Erlang nodes, usually located in single locations. Anything more complex will need to be left to the developers using one of the following methods:

- Switching to SSL
- Implementing their own high-level communication layer
- Tunneling over secure channels
- Reimplementing the communication protocol between nodes

Using SSL is explained in the Secure Socket Layer User's Guide (Chapter 3, "Using SSL for Erlang Distribution," at *http://www.erlang.org/doc/apps/ssl/ssl_distribution.html*). Pointers on how to implement your own carrier protocol for the Erlang distribution are provided in the ERTS User's Guide (Chapter 3, "How to implement an alternative carrier for the Erlang distribution" at *http://www.erlang.org/doc/apps/erts/alt_dist.html*), which also contains details on the distribution protocol (Chapter 9, "Distribution Protocol," at *http://www.erlang.org/doc/apps/erts/erl_dist_protocol.html*).

Even in these cases, you must be careful, because someone gaining access to one of the distributed nodes then has access to all of them, and can run any command those nodes can.

Topology Doesn't Change

When first designing a distributed application made to run on many servers, it is possible that you will have a given number of servers in mind, and perhaps a given list of hostnames. Maybe you will design things with specific IP addresses in mind. This can be a mistake. Hardware dies, operations people move servers around, new machines are added, and some are removed. The topology of your network will constantly change. If your application works with any of these topological details hardcoded, then it won't easily handle these kinds of changes in the network.

In the case of Erlang, there is no explicit assumption made in that way. However, it is very easy to let it creep inside your application. Erlang nodes all have a name and a hostname, and they can constantly be changing. With Erlang processes, you not only need to think about how the process is named, but also about where it is now located in a cluster. If you hardcode both the names and hostnames, you might be in trouble at the next failure. Don't worry too much though. As discussed in "The Calls from Beyond" on page 467, we can use a few interesting libraries that let us forget about node names and topology in general, while still being able to locate specific processes.

There Is Only One Administrator

This fallacy is something a distribution layer of a language or library can't prepare you for, no matter what. The idea of this fallacy is that you do not always have only one main operator for your software and its servers, although it might be designed as if there were only one. If you decide to run many nodes on a single computer, then you might never need to think about this fallacy. However, if you run stuff across different locations, or a third party depends on your code, then you must take care.

Things to pay attention to include giving others tools to diagnose problems on your system. Erlang is somewhat easy to debug when you can manipulate a VM manually—you can even reload code on the fly if you need to, after all. Someone who cannot access your terminal and sit in front of the node will need different facilities to operate though.

Another aspect of this fallacy is that things like restarting servers, moving instances between data centers, and upgrading parts of your software stack are not necessarily controlled by a single person or team. In very large software projects, it is likely that many teams, or even many different software companies, take charge of different parts of a greater system.

If you're writing protocols for your software stack, being able to handle many versions of that protocol might be necessary, depending on how fast or slow your users and partners are to upgrade their code. The protocol might contain information about its versioning from the beginning, or be able to change halfway through a transaction, depending on your needs. I'm sure you can think of more examples of things that can go wrong.

Transport Cost Is Zero

The transport cost is zero fallacy is a two-sided assumption. The first one relates to the cost of transporting data in terms of time, and the second one is related to the cost of transporting data in terms of money.

The first case assumes that doing things like serializing data is nearly free, very fast, and doesn't play a big role. In reality, larger data structures take longer to be serialized than small ones, and then they need to be unserialized on the other end of the wire. This will be true no matter what you carry across the network, although small messages help reduce the noticeability of this effect.

The second aspect of assuming transport cost is zero has to do with how much it costs to carry data around. In modern server stacks, memory (both in RAM and on disk) is often cheap compared to the cost of bandwidth, which you need to pay for continuously, unless you own the whole network where things run. Optimizing for fewer requests with smaller messages will be rewarding in this case and many others, too.

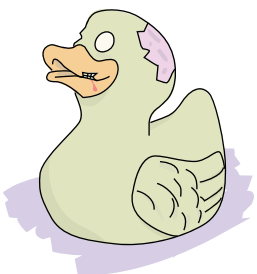
For Erlang, due to its initial use cases, no special care has been taken to do things like compress messages going cross-node (although the functions for it already exist). Instead, the original designers chose to let people implement their own communication layer if they required it. The responsibility is thus on the programmer to make sure small messages are sent and other measures are taken to minimize the costs of transporting data.

The Network Is Homogeneous

The last assumption is thinking that all components of a networked application will speak the same language or will use the same formats to operate together.

For our zombie survivors, this can be a question of not assuming that all survivors will always speak English (or good English) when they lay their plans, or that a word will have the same meaning to different people.

In terms of programming, this is usually about not relying on closed standards but using open ones instead, or being ready to switch from one protocol to another one at any point in time. When it comes to Erlang, the distribution protocol is entirely public, but all Erlang nodes assume that people communicating with them speak the same language. Foreigners trying to integrate themselves into an Erlang cluster either must learn to speak Erlang's protocol or Erlang applications need some kind of translation layer for XML, JSON, or whatever.



Learning to speak Erlang's protocol is relatively simple. If you respect the protocol, you can pretend to be another Erlang node, even if you're not writing Erlang. If it quacks like a duck and walks like a duck, then it must be a duck. That's the principle behind *C nodes*. C nodes (or nodes in languages other than C) are programs that implement Erlang's protocol and then pretend they are Erlang nodes in a cluster, allowing you to distribute work without too much effort. More details are available at *http://www.erlang.org/doc/tutorial/cnode.html*.

Another solution for data exchange is to use BERT or BERT-RPC (documented at *http://bert-rpc.org/*). This is an exchange format like XML or JSON, but specified as something similar to the Erlang *external term format* (documented at *http://www.erlang.org/doc/apps/erts/erl_ext_dist.html*).

Fallacies in a Nutshell

We've looked at a bunch of assumptions involved in distributed computing fallacies. In short, you always need to be careful and consider the following points:

- You shouldn't assume the network is reliable. Erlang doesn't have any special measures for that, except detecting that something went wrong for you (although that's not too bad as a feature).
- The network might be slow from time to time. Erlang provides asynchronous mechanisms and knows about it, but you need to be careful that your application is also aware of this possibility.
- Bandwidth isn't infinite. Small, descriptive messages help respect this.
- The network isn't secure, and Erlang doesn't have anything to offer by default for this.
- The topology of the network can change. No explicit assumption is made by Erlang, but you might make some assumptions about where things are and how they're named.
- You (or your organization) rarely fully control the structure of things. Parts of your system may be outdated, use different versions, or be restarted or down when you don't expect that to happen.
- Transporting data has costs. Again, small, short messages help.
- The network is heterogeneous. Not everything is the same, and data exchange should rely on well-documented formats.

Dead or Dead-Alive

The fallacies of distributed computing partially explain why we're fighting monsters in the dark. Although we have some useful tools, there are still a lot of issues and things left for us to do. We need to be careful about design decisions (small messages, reducing communication, and so on) regarding these fallacies. The most problematic issue has to do with nodes dying or the network being unreliable. This issue is especially nasty because there is no good way to know whether something is dead or alive (without being able to contact it).

Let's get back to Bill, Zoey, Rick, and Daryl, our four zombie apocalypse survivors. They all met at a safe house, spent a few days resting there, eating whatever canned food they could find. After a while, they needed to move out and split across town to find more resources. They've set a rendezvous point in a small camp on the limits of the town they're in. During the expedition, they keep contact by talking with the walkie-talkies. They announce what they found, such as other survivors.

Now suppose that at some point between the safe house and the rendezvous point, Rick tries to contact his comrades. He manages to call Bill and Zoey, but Daryl isn't reachable. Bill and Zoey can't contact him either. The problem is that there is absolutely no way to know if Daryl has been devoured by zombies, if his battery is dead, if he is asleep, or if he is just underground. The group must decide whether to keep waiting for him, keep calling for a while, or assume he is dead and move forward.

The same dilemma exists with nodes in a distributed system. When a node becomes unresponsive, is it gone because of a hardware failure? Did the application crash? Is there congestion on the network? Is the network down? In some cases, the application is not running anymore, and you can simply ignore that node and continue what you're doing. In other cases, the application is still running on the isolated node; from that node's perspective, everything else is dead.

Erlang made the default decisions of considering unreachable nodes as dead nodes and reachable nodes as alive. This is a pessimistic approach that makes sense if you want to quickly react to catastrophic failures. It assumes that the network is generally less likely to fail than the hardware or the software in the system, which makes sense considering how Erlang was used originally. An optimistic approach (which assumes nodes are still alive) could delay crash-related measures because it assumes that the network is more likely to fail than hardware or software, and thus have the cluster wait longer for the reintegration of disconnected nodes.

This raises a question: In a pessimistic system, what happens when the node we thought was dead suddenly returns, and it turns out it never died? We're caught by surprise by a living dead node, which had a life of its own, isolated from the cluster in every way—data, connections, and so on. Some very annoying things can happen.

Let's imagine for a moment that you have a system with two nodes in two different data centers. In that system, users have money in their account, with the full amount held on each node. Each transaction then synchronizes the data with all the other nodes. When all the nodes are fine, users can keep spending money until their account is empty.

The software is chugging along fine, but at some point, one of the nodes gets disconnected from the other. There is no way to know if the other side is alive or dead. For all we care, both nodes could still be receiving requests from the public, but without being able to communicate with each other.

There are two general strategies that can be taken: stop all transactions or don't. The risk of picking the first one is that your product becomes unavailable and you're losing money. The risk of the second one is that a user with $1,000 in his account now has two servers that can accept $1,000 of transactions, for a total of $2,000! Whatever we do, we risk losing money if we don't do things right.

Isn't there a way to avoid the issue entirely by keeping the application available during netsplits, without losing data in between servers?

My Other Cap Is a Theorem

A quick answer to the previous question is no. There is sadly no way to keep an application alive and correct at the same time during a netsplit.

This idea is known as the *CAP theorem*. The CAP theorem first states that there are three core attributes to all distributed systems: **c**onsistency, **a**vailability, and **p**artition tolerance.

Consistency

In the previous example, consistency would be having the ability to have the entire system, whether there are 2 or 1,000 nodes that can answer queries, see exactly the same amount of money in the account at a given time. This is something usually done by adding transactions (where all nodes must agree to making a change to a database as a single unit before doing so) or some other equivalent mechanism.

By definition, the idea of consistency is that all operations look as if they were completed as a single, indivisible block, even across many nodes. This is not in terms of time, but in terms of not having two different operations on the same piece of data modifying them in ways that give multiple values reported by system during these operations. It should be possible to modify a piece of data and not need to worry about others ruining your day by fiddling with it at the same time you are.

Availability

The idea behind availability is that if you ask the system for some piece of data, you're able to get a response. If you don't get an answer back, the system isn't available to you. Note that a response that says, "Sorry, I can't figure out results because I'm dead," isn't really a response, but only a sad excuse for one. There is no more useful information in this response than in no response at all (although academics are somewhat divided on the issue).

NOTE *An important consideration in the CAP theorem is that availability is a concern only to nodes that are* not dead. *A dead node cannot send responses because it can't receive queries in the first place. This isn't the same as a node that can't send a reply because a thing it depends on is no longer there. If the node can't take requests, change data, or return erroneous results, it isn't technically a threat to the balance of the system in terms of correctness. The rest of the cluster just needs to handle more of the load until the dead node comes back up and can be synchronized.*

Partition Tolerance

Partition tolerance is the tricky part of the CAP theorem. It usually means that the system can keep on working (and contain useful information) even when parts of it can no longer communicate with each other. The whole point of partition tolerance is that the system can work with messages possibly being lost between components. The definition is a bit abstract and open-ended, and we'll see why.

The CAP theorem basically specifies that in any distributed system, you can have only two of the CAP attributes: consistency + availability (CA), consistency + partition tolerance (CP), or availability + partition tolerance (AP). There is no possible way to have all of them. This is both bad and good news. The bad news is that it's impossible to have both consistency and availability while the network is partitioned. The good news is that this is a theorem. If customers ask you to provide all three of them, you will have the advantage of being able to tell them it is literally impossible to do so, and won't need to lose too much time (outside of explaining to them what the hell the CAP theorem is).

Of the three possibilities, one that we can usually dismiss is the idea of CA. The only time you would really want this is if you dare to say the network will never fail, or that if it does, it does as an atomic unit (if one thing fails, everything fails at once).

Until someone invents a network and hardware that never fail, or has some way to make all parts of a system fail at once if one of them does, failure must be an option. Only two combinations of the CAP theorem remain: AP or CP. A system torn apart by a netsplit can either remain available or consistent, but not both.

NOTE *Some systems will choose to have neither A nor C. In some cases of high performance, criteria such as throughput (how many queries you can answer) or latency (how fast you can answer queries) will bend things in a way such that the CAP theorem isn't about two attributes (CA, CP, or AP), but also about two or fewer attributes. Also note that some system can be fully consistent when there is no netsplit but relax consistency requirements when netsplits happen.*

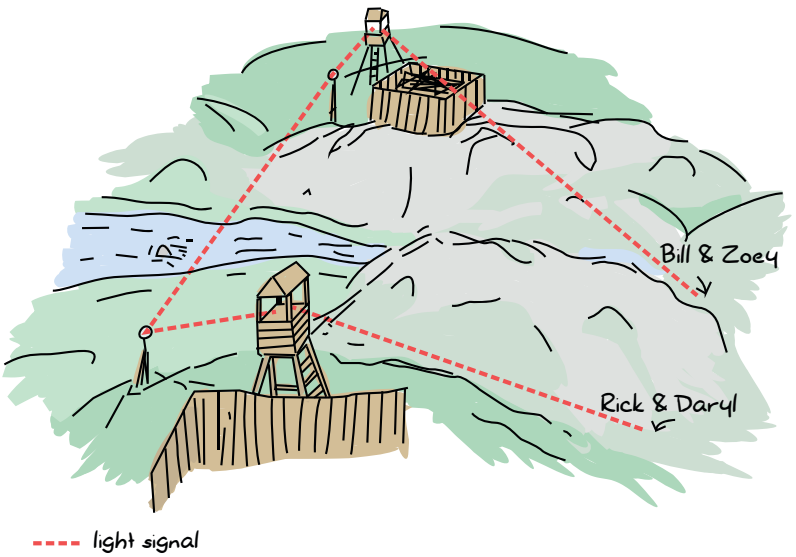
Zombie Survivors and CAP

Time passed for our group of survivors, and they fended off groups of undead for a good while. Bullets pierced brains, baseball bats shattered skulls, and infected people were left behind. Bill, Zoey, Rick, and Daryl's batteries eventually ran out, and they were unable to communicate. As luck would have it, they all found two survivor colonies populated with computer

scientists and engineers enamored with zombie survival. The colony survivors were familiar with the concepts of distributed programming, as well as communicating using light signals and mirrors with homemade protocols.

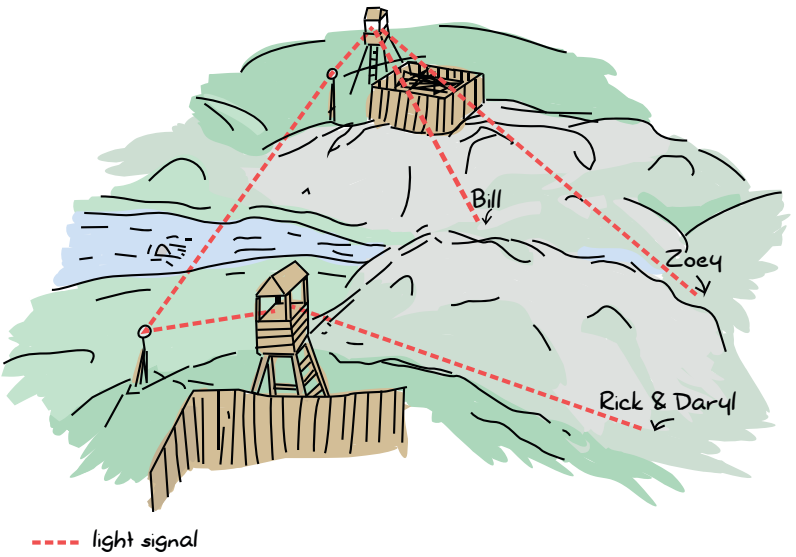
Bill and Zoey found the Chainsaw colony, and Rick and Daryl found the Crossbow camp. Given that our survivors were the newest arrivals in their respective colonies, they were often delegated to go out in the wild, hunt for food, and kill zombies coming too close to the perimeters, while the rest of the people debated the merits of Vim versus Emacs (the only war that couldn't die after a complete zombie apocalypse).

On their hundredth day there, our four survivors were sent to meet halfway across the camps to trade goods for each colony. Before the survivors left for their meeting, the Chainsaw and Crossbow colonies agreed on a rendezvous point. If the destination or meeting time were to change while they were on their journey, Rick and Daryl could send a message to the Crossbow colony, or Zoey and Bill could send a message to the Chainsaw colony. Then each colony would communicate the information to the other colony, which would forward the changes to the other survivors, as shown here:



All four survivors left early on a Sunday morning for a long trip on foot, due to meet on Friday morning before dawn. Everything went fine (except the occasional skirmishes with dead people who had been alive for quite a while at this point).

Unfortunately, on Wednesday, due to heavy rain and increased zombie activity, Bill and Zoey were separated, lost, and delayed. The new situation looked a bit like this:



To make matters worse, after the rain, the usually clear sky between the two colonies got foggy, and it became impossible for the Chainsaw computer scientists to communicate with the Crossbow people. Bill and Zoey communicated their problems to their colony and asked to set a new meeting time. This would have been all right without the fog, but now they have the equivalent of a netsplit.

If both camps work under the CP approach, they will just keep Zoey and Bill from setting a new meeting time. The CP approach is usually all about stopping modifications to the data so it remains consistent, and all survivors can still ask their respective camps for the data from time to time. They will just be denied the right to change it. This ensures that there is no way for some survivors to mess up the planned meeting time; any other survivors cut off from any contact could still meet at that time.

If both camps pick AP instead of CP, then survivors could be allowed to change meeting dates. Each side of the partition would have its own version of the meeting data. So if Bill called for a new meeting for Friday night, the general state becomes as follows:

```
Chainsaw: Friday night
Crossbow: Friday before dawn
```

As long as the split lasts, Bill and Zoey will get their information from Chainsaw only, and Rick and Daryl from Crossbow only. This lets some of the survivors reorganize themselves if needed.

The interesting problem here is how to handle the different versions of the data when the split is resolved (and the fog goes away). The CP approach to this is pretty straightforward: The data didn't change, so there is nothing to do. The AP approach has more flexibility but also more problems to resolve. Usually, different strategies are employed:

- Use the *last write wins* approach, which is a conflict resolution method that keeps the last update that was made. This one can be tricky because in distributed settings, timestamps can be off, or things can happen at exactly the same time.
- A winner can be picked randomly.
- Use more sophisticated time-based methods to help reduce conflicts, such as last write wins, but with logical clocks. Logical clocks do not work with absolute time values, but with incrementing values every time someone modifies a file. If you want to know more about this approach, read up on *Lamport clocks* or *vector clocks*.
- The onus of picking what to do with the conflict can be pushed back to the application (or in our case, to the survivors). The receiving end will just need to choose which of the conflicting entries is the right one. This is a bit like what happens when you have merge conflicts with source control tools like Subversion, Mercurial, or Git.

Which one is better? The way I've described things may have led you to believe that we have the choice to be either fully AP or fully CP, like an on/off switch. In the real world, we can have various options, such as a quorum system, which changes this yes/no question into a dial we can turn to choose how much consistency we want.

A quorum system works by a few simple rules. You have N nodes in the system and require M of them to agree to modify the data to make it possible. A system with a relatively low consistency requirement could ask for only 15 percent of the nodes to be available to make a change. This means that in cases of splits, even small fragments of the network are still able to modify the data. A higher consistency rating, set to maybe 75 percent of the nodes, would mean that a larger part of the system needs to be present in order to make changes. In this situation, if a few of the nodes are isolated, they won't have the right to change the data. However, the major part of the system that's still interconnected can work fine.

By making the M value of required nodes equal to N (the total number of nodes), you can have a fully consistent system. By giving M the value 1, you have a fully AP system, with no consistency guarantees.

Moreover, you could play with these values on a per-query basis. Queries having to do with things of little importance (someone just logged on!) can have lower consistency requirements, while things having to do with inventory and money could require more consistency. Mix this in with different conflict resolution methods for each case, and you can build surprisingly flexible systems.

Combined with all the different conflict resolution solutions available, a lot of options become available to distributed systems, but their implementation remains very complex. We won't use them in detail, but I think it's important to know what's available out there and to be aware of the different options.

Now that we've looked at the issues involved with distributed computing with Erlang, let's turn to the specifics of setting up a basic distributed system.

Setting Up an Erlang Cluster

Except for handling the fallacies of distributed computing, the hardest part of distributed Erlang is managing to set up things right. Connecting nodes across different hosts is a special kind of pain. To avoid this, we'll usually try things out using many nodes on a single computer, which tends to make things easier.

Through the Desert on a Node with No Name

As mentioned earlier, Erlang gives names to each of the nodes to be able to locate and contact them. The names are of the form `Name@Host`, where the host is based on available DNS entries, either over the network or in your computer's host files (*/etc/hosts* on Mac OS X, Linux, and other Unix-like systems, and *C:\Windows\system32\drivers\etc\hosts* for most Windows installations). All names need to be unique to avoid conflicts. If you try to start a node with the same name as another one on the same hostname, you'll get a pretty terrible crash message.

Before starting any nodes, we need to talk a bit about their names. There are two types of names:

- *Long names* are based on fully qualified domain names (*aaa.bbb.ccc*). Many DNS resolvers consider domain names to be fully qualified if they contain a period (.).
- *Short names* are based on hostnames without a period. They are resolved by going through your host file or through any possible DNS entry.

It is generally easier to set up a bunch of Erlang nodes on a single computer using short names rather than long names.

Also note that nodes with short names cannot communicate with nodes that have long names, and the opposite is also true.

To use short names, start the Erlang VM with `erl -sname short_name@domain`. To use long names start the VM with `erl -name long_name@some.domain`. You can also start nodes with only the names: `erl -sname short_name` or `erl -name long_name`. Erlang will automatically attribute a hostname based on your operating system's configuration. Another option is to start a node and specify a direct IP address, such as `erl -name name@127.0.0.1`.

NOTE *Windows users should still use* `werl` *instead of* `erl`*. However, in order to start a distributed node and give it a name, the node should be started from the command line instead of by clicking some shortcut or executable.*

Connecting Nodes

Let's start two nodes:

```
$ erl -sname ketchup
... <snip> ...
(ketchup@ferdmbp)1>
```

```
$ erl -sname fries
... <snip> ...
(fries@ferdmbp)1>
```

To connect `fries` with `ketchup` (and make a delicious cluster), go to the first shell and enter the following function:

```
(ketchup@ferdmbp)1> net_kernel:connect(fries@ferdmbp).
true
```

The `net_kernel:connect(NodeName)` function sets up a connection with another Erlang node. (Some tutorials use `net_adm:ping(Node)`, but I think `net_kernel:connect/1` sounds more serious and lends me credence!) If you see `true` as the result from the function call, congratulations, you're in distributed Erlang mode now. If you see `false`, you're in for a world of hurt trying to get your network to play nice. For a very quick fix, edit your host files to accept whatever host you want to use. Then try again.

You can see your own node name by calling the BIF `node()` and see who you're connecting to by calling the BIF `nodes()`:

```
(ketchup@ferdmbp)2> node().
ketchup@ferdmbp
(ketchup@ferdmbp)3> nodes().
[fries@ferdmbp]
```

To get the nodes to communicate with each other, we'll try a simple trick. Register each shell's process as `shell` locally:

```
(ketchup@ferdmbp)4> register(shell, self()).
true
```

```
(fries@ferdmbp)1> register(shell, self()).
true
```

Now you'll be able to call the process by name by sending a message to `{Name, Node}`. Let's try this on both shells:

```
(ketchup@ferdmbp)5> {shell, fries@ferdmbp} ! {hello, from, self()}.
{hello,from,<0.52.0>}
```

```
(fries@ferdmbp)2> receive {hello, from, OtherShell} -> OtherShell ! <<"hey there!">> end.
<<"hey there!">>
```

So that message was apparently received, and we can send something else to the other shell:

```
(ketchup@ferdmbp)6> flush().
Shell got <<"hey there!">>
ok
```

As you can see, we can transparently send tuples, atoms, pids, and binaries without a problem. Any other Erlang data structure is fine, too.

And that's it. You know how to work with distributed Erlang!

More Tools

Several other BIFs might be useful for working with distributed Erlang. The `erlang:monitor_node(NodeName, Bool)` function lets the process that calls it with `true` as a value for `Bool` receive a message of the format `{nodedown, NodeName}` if the node dies.

NOTE *Unless you're writing a special library that relies on checking the life of other nodes, you will rarely need to use `erlang:monitor_node/2`. This is because functions like `link/1` and `monitor/2` still work across nodes. However, it may be interesting to use `erlang:monitor_node/2` when you have a lot of monitors or links across many nodes. If you used the usual `link/1` and `monitor/2` functions in that case, a node dying could mean that thousands of monitors would fire at once, instead of only one event for the node monitor, which can then relay information locally.*

Suppose we set up the following from the `fries` node:

```
(fries@ferdmbp)3> process_flag(trap_exit, true).
false
(fries@ferdmbp)4> link(OtherShell).
true
(fries@ferdmbp)5> erlang:monitor(process, OtherShell).
#Ref<0.0.0.132>
```

Then we kill the ketchup node. In this case, the fries shell process should receive an 'EXIT' and monitor message:

```
(fries@ferdmbp)6> flush().
Shell got {'DOWN',#Ref<0.0.0.132>,process,<6349.52.0>,noconnection}
Shell got {'EXIT',<6349.52.0>,noconnection}
ok
```

And that's the kind of stuff you'll see.

NOTE *Instead of killing a node to disconnect it, you may also want to try the BIF `erlang:disconnect_node(Node)` to get rid of the node without shutting it down.*

But hey, wait a minute! Why does the pid look like that? Are we seeing things right?

```
(fries@ferdmbp)7> OtherShell.
<6349.52.0>
```

What? Shouldn't this be <0.52.0>? Nope. See, that way of displaying a pid is just some kind of visual representation of what a process identifier is really like. The first number represents the node (where 0 means the process is coming from the current node), the second one is a counter, and the third is a second counter for when you have so many processes created that the first counter is not enough. The true underlying representation of a pid is more like this:

```
(fries@ferdmbp)8> term_to_binary(OtherShell).
<<131,103,100,0,15,107,101,116,99,104,117,112,64,102,101,
  114,100,109,98,112,0,0,0,52,0,0,0,0,3>>
```

The binary sequence <<107,101,116,99,104,117,112,64,102,101,114,100,109,98,112>> is in fact a Latin-1 (or ASCII) representation of <<"ketchup@ferdmbp">>, the name of the node where the process is located. Then we have the two counters: <<0,0,52>> and <<0,0,0,0>>. The last value (3) is some token value to differentiate whether the pid comes from an old node, a dead one, and so on. That's why pids can be used transparently anywhere.

NOTE *If you're unsure which node a pid is coming from, you don't need to convert it to a binary to read the node name. Just call `node(Pid)`, and the node where it's running will be returned as an atom.*

Other interesting BIFs to use are `spawn/2`, `spawn/4`, `spawn_link/2`, and `spawn_link/4`. They work like the other `spawn` BIFs, except these let you spawn functions on remote nodes. Try this from the `ketchup` node:

```
(ketchup@ferdmbp)6> spawn(fries@ferdmbp,
(ketchup@ferdmbp)6>       fun() -> io:format("I'm on ~p~n", [node()]) end).
I'm on fries@ferdmbp
<6448.50.0>
```

This is essentially a remote procedure call. We can choose to run arbitrary code on other nodes, without giving ourselves more trouble than that! Interestingly, the function is running on the other node, but we receive the output locally. That's right—even output can be transparently redirected across the network. This is possible thanks to the idea of group leaders. Group leaders are inherited in the same way whether they're local or remote and will forward IO operations to their parents until they hit the correct output driver, in the original shell that called them.

Those are all the tools you need in Erlang to be able to write distributed code. You have just received your machete, flashlight, and mustache. You're at a level that would take a long while to achieve with other languages without such a distribution layer. Now is the time to kill monsters. Or maybe first, we need to talk about the cookie monster.

Cookies

Earlier in the chapter, I illustrated how Erlang node connections are set up as meshes. If someone connects to a node, it gets connected to all the other nodes. However, you may want to run different Erlang node clusters on the same piece of hardware. In this case, you do not want to accidentally connect two Erlang node clusters together. To help with this, the designers of Erlang added a little token value called a *cookie*.

Many references, such as the official Erlang documentation, put cookies under the topic of security. But that has to be seen as a joke, because there's no way anyone seriously considers cookies to be safe things.

The cookie is a little unique value that must be shared between nodes to allow them to connect with each other. Cookies are closer to the idea of usernames than passwords, and I'm pretty sure no one would consider having a username (and nothing else) as a security feature. Cookies make much more sense as a mechanism used to divide clusters of nodes than as an authentication mechanism.

To give a cookie to a node, just start it by adding a `-setcookie Cookie` argument to the command line. Let's try this with two new nodes:

```
$ erl -sname salad -setcookie 'myvoiceismypassword'
... <snip> ...
(salad@ferdmbp)1>
```

```
$ erl -sname mustard -setcookie 'opensesame'
... <snip> ...
(mustard@ferdmbp)1>
```

Now both nodes have different cookies, and they shouldn't be able to communicate:

```
(salad@ferdmbp)1> net_kernel:connect(mustard@ferdmbp).
false
```

This one has been denied, but we don't see many explanations as to why. Let's take a look at the `mustard` node:

```
=ERROR REPORT==== 10-Dec-2013::13:39:27 ===
** Connection attempt from disallowed node salad@ferdmbp **
```

Good. Now what if we did really want salad and mustard to be together? There's a BIF called `erlang:set_cookie/2` to do what we need. If you call `erlang:set_cookie(OtherNode, Cookie)`, you will use that cookie only when connecting to that other node. If you instead use `erlang:set_cookie(node(), Cookie)`, you'll be changing the node's current cookie for all future connections. To see the changes, use `erlang:get_cookie()`:

```
(salad@ferdmbp)2> erlang:get_cookie().
myvoiceismypassword
(salad@ferdmbp)3> erlang:set_cookie(mustard@ferdmbp, opensesame).
true
(salad@ferdmbp)4> erlang:get_cookie().
myvoiceismypassword
(salad@ferdmbp)5> net_kernel:connect(mustard@ferdmbp).
true
(salad@ferdmbp)6> erlang:set_cookie(node(), now_it_changes).
true
(salad@ferdmbp)7> erlang:get_cookie().
now_it_changes
```

There is one last cookie mechanism to see. If you tried the earlier examples in this chapter, go look in your home directory. You should see a file named *.erlang.cookie* in there. If you read that file, you'll see a random

string that looks a bit like PMIYERCHJZNZGSRJPVRK. Whenever you start a distributed node without a specific command to give it a cookie, Erlang will create one and put it in *.erlang.cookie*. Then every time you start a node again without specifying its cookie, the VM will look into your home directory and use whatever is in that file.

Remote Shells

One of the first things we've learned in Erlang was how to interrupt running code by using CTRL-G (^G). In there, we saw a menu for distributed shells:

```
(salad@ferdmbp)1>
User switch command
--> h
c [nn]            - connect to job
i [nn]            - interrupt job
k [nn]            - kill job
j                 - list all jobs
s [shell]         - start local shell
r [node [shell]]  - start remote shell
q        - quit erlang
? | h             - this message
```

The `r [node [shell]]` option is the one we're looking for to work with our remote shells. We can start a job on the `mustard` node as follows:

```
--> r mustard@ferdmbp
--> j
  1  {shell,start,[init]}
  2* {mustard@ferdmbp,shell,start,[]}
--> c
Eshell V5.9.1  (abort with ^G)
(mustard@ferdmbp)1> node().
mustard@ferdmbp
```

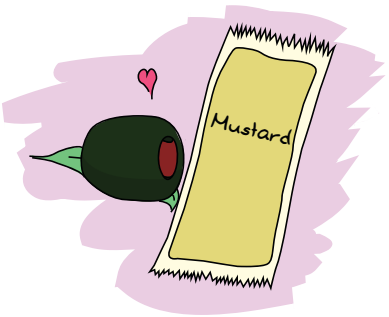
And there you have it. You can now use the remote shell the same way you would use a local one. There are a few differences with older versions of Erlang, where features like autocompletion are not available. This way of doing things is still very useful whenever you need to change things on a node running with the `-noshell` option. If the `-noshell` node has a name, then you can connect to it to do DevOps-related things like reloading modules, debugging some code, and so on.

By using CTRL-G again, you can go back to your original node. Be careful when you stop your session though. If you call `q()` or `init:stop()`, you'll be terminating the remote node!

Hidden Nodes

Erlang nodes can be connected by calling `net_kernel:connect/1`, but you need to be aware that pretty much any interaction between nodes will get them to set up a connection. Calling `spawn/2` or sending a message to a foreign pid will automatically set up connections.

This might be rather annoying if you have a decent cluster and you want to communicate with a single node to change a few things there. You wouldn't want your admin node to suddenly be integrated into the cluster, making other nodes believe that they have a new coworker to send tasks to. To connect to a remote node without automatically connecting to all the other nodes it is connected to, you could call the rarely used `erlang:send(Dest, Message, [noconnect])` function, which sends a message without creating a connection, but this is rather error prone.



Instead, you want to set up a node with the `-hidden` flag.

Let's say you're still running the `mustard` and `salad` nodes. We'll start a third node, `olives`, which will connect only to `mustard` (make sure the cookies are the same!):

```
$ erl -sname olives -hidden
... <snip> ...
(olives@ferdmbp)1> net_kernel:connect(mustard@ferdmbp).
true
(olives@ferdmbp)2> nodes().
[]
(olives@ferdmbp)3> nodes(hidden).
[mustard@ferdmbp]
```

Aha! The node didn't connect to `ketchup`, and at first sight, it didn't appear to connect with `mustard` either. However, calling `nodes(hidden)` shows that we do have a connection there! Let's see what the `mustard` node sees:

```
(mustard@ferdmbp)1> nodes().
[salad@ferdmbp]
(mustard@ferdmbp)2> nodes(hidden).
[olives@ferdmbp]
(mustard@ferdmbp)3> nodes(connected).
[salad@ferdmbp,olives@ferdmbp]
```

This is a similar view, but now we add the `nodes(connected)` BIF that shows all connections, regardless of their type. The `ketchup` node will never see any connection to `olives`, unless especially told to connect there.

One other interesting use of `nodes/1` is calling `nodes(known)`, which will show all nodes that the current node ever connected to.

With remote shells, cookies, and hidden nodes, managing a distributed Erlang system becomes simpler.

The Walls Are Made of Fire, and the Goggles Do Nothing

If you need to go through a firewall with distributed Erlang (and do not want to tunnel), you will likely want to open a few ports here and there for Erlang communication. In this case, you should open port 4369, the default port for EPMD (the Erlang Port Mapper Daemon application introduced earlier in the chapter). It's a good idea to use this port, because it has been officially registered for EPMD by Ericsson. This means that any standards-compliant operating system you use will have that port free, ready for EPMD.

Then you will want to open a range of ports for connections between nodes. The problem is that Erlang just assigns random port numbers to inter-node connections. There are, however, two hidden application variables that let you specify a range within which ports can be assigned. The two values are `inet_dist_listen_min` and `inet_dist_listen_max` from the `kernel` application.

You could, as an example, start Erlang as `erl -name left_4_distribudead -kernel inet_dist_listen_min 9100 -kernel inet_dist_listen_max 9115` in order to set a range of 15 ports to be used for Erlang nodes. Alternatively, you could specify these ports with a configuration file named *ports.config* that looks a bit like this:

```
[{kernel,[
  {inet_dist_listen_min, 9100},
  {inet_dist_listen_max, 9115}
]}].
```

And then starting the Erlang node as `erl -name the_army_of_darknodes -config ports`. The variables will be set in the same way. Note that these ports are listen ports, so you need to keep only one per node per machine. If you are running two VMs on a given server or computer, you'll need two listen ports.

The Calls from Beyond

On top of all the BIFs and options we've covered so far, there are a few modules that can be used to help developers work with distribution.

The net_kernel Module

`net_kernel` is the module we used to connect and disconnect nodes earlier.

It has some other fancy functionality, such as being able to transform a nondistributed node into a distributed one:

```
$ erl
... <snip> ...
1> net_kernel:start([romero, shortnames]).
{ok,<0.43.0>}
(romero@ferdmbp)2>
```

You can use either `shortnames` or `longnames` to define whether you want to have the equivalent of `-sname` or `-name`. Moreover, if you know a node is going to be sending large messages and thus might need a long heartbeat time between nodes, a third argument can be passed to the list: `net_kernel:start([Name, Type, HeartbeatInMilliseconds])`. By default, the heartbeat delay (also called *tick time*) is set to 15 seconds, or 15,000 milliseconds.

Two other functions of the module are `net_kernel:set_net_ticktime(S)`, which lets you change the tick time of the node to avoid disconnections, and `net_kernel:stop()`, which switches a node from being distributed back to being a normal node:

```
(romero@ferdmbp)2> net_kernel:set_net_ticktime(5).
change_initiated
(romero@ferdmbp)3> net_kernel:stop().
ok
4>
```

The global Module

The next useful module for distribution is `global`. The `global` module is an alternative process registry. It automatically spreads its data to all connected nodes, replicates data there, handles node failures, and supports different conflict-resolution strategies when nodes get back online.

You register a name by calling `global:register_name(Name, Pid)`, and unregister with `global:unregister_name(Name)`. In case you want to do a name transfer without ever having it point to nothing, you can call `global:re_register_name(Name, Pid)`. You can find a pid with `global:whereis_name(Name)`, and send a message to a process by calling `global:send(Name, Message)`. There

is everything you need. What's especially nice is that the names you use to register the processes can be *any* terms at all.

A naming conflict will happen when two nodes suddenly connect and both have two different processes sharing the same name. In these cases, `global` will kill one of them randomly by default. There are ways to override that behavior. Whenever you register or reregister a name, pass a third argument to the function:

```
5> Resolve = fun(_Name,Pid1,Pid2) ->
5>     case process_info(Pid1, message_queue_len) > process_info(Pid2, message_queue_len) of
5>         true -> Pid1;
5>         false -> Pid2
5>     end
5> end.
#Fun<erl_eval.18.59269574>
6> global:register_name({zombie, 12}, self(), Resolve).
yes
```

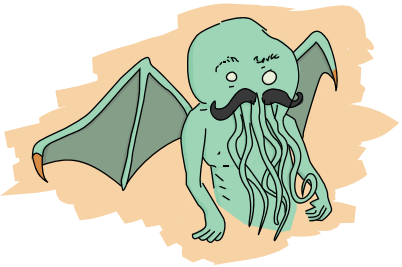
The `Resolve` function will pick the process with the most messages in its mailbox as the one to keep (it's the one the function returns the pid of). You could alternatively contact both processes and ask for who has the most subscribers, or keep only the first one to reply, to name a few strategies you could implement. If the `Resolve` function crashes or returns something other than the pids, the process name is unregistered. For your convenience, the `global` module already defines three functions:

1. `fun global:random_exit_name/3` kills a process randomly. This is the default option.
2. `fun global:random_notify_name/3` randomly picks one of the two processes as the one to survive, and it will send `{global_name_conflict, Name}` to the process that lost.
3. `fun global:notify_all_name/3` unregisters both pids and sends the message `{global_name_conflict, Name, OtherPid}` to both processes. It lets them resolve the issue themselves so they reregister again.

The `global` module has one downside in that it is often said to be rather slow to detect name conflicts and nodes going down. It's also better for a somewhat small number of registrations that tend not to change too much over time. Other than these limitations, `global` is a fine module, and it's even supported by behaviors. Just change all the `gen_Something:start_link(...)` calls that use local names (`{local, Name}`) to instead use `{global, Name}`, and then all calls and casts (and their equivalents) to use `{global, Name}` instead of just `Name`. When you do this, things will be distributed.

NOTE *Erlang versions from R15B01 and newer allow the usage other registries than `local` and `global`. Supply the name as `{via, RegistryModule, Name}` to use whatever compatible process registry you want for your processes.*

The rpc Module



The next module on the list is rpc (for remote procedure call). It contains functions that let you execute commands on remote nodes, as well as a few functions that facilitate parallel operations. To test these out, let's begin by starting two different nodes and connecting them together, as demonstrated earlier in the chapter. Name the nodes cthulu and lovecraft.

The most basic rpc operation is rpc:call/4-5. It allows you to run a given operation on a remote node and get the results locally:

```
(cthulu@ferdmbp)1>  rpc:call(lovecraft@ferdmbp, lists, sort, [[a,e,f,t,h,s,a]]).
[a,a,e,f,h,s,t]
(cthulu@ferdmbp)2>  rpc:call(lovecraft@ferdmbp, timer, sleep, [10000], 500).
{badrpc,timeout}
```

As seen in this Call of the Cthulu node, the function with four arguments takes the form rpc:call(Node, Module, Function, Args). Adding a fifth argument gives a timeout. The rpc call will return whatever was returned by the function it ran, or {badrpc, Reason} in case of a failure.

If you've studied some distributed or parallel computing concepts before, you might have heard of *promises* or *futures*. Promises and futures are a bit like remote procedure calls, except that they are asynchronous. The rpc module lets us have this:

```
(cthulu@ferdmbp)3> Key = rpc:async_call(lovecraft@ferdmbp, erlang, node, []).
<0.45.0>
(cthulu@ferdmbp)4> rpc:yield(Key).
lovecraft@ferdmbp
```

By combining the result of the function rpc:async_call/4 with the function rpc:yield(Res), we can have asynchronous remote procedure calls and fetch the results later. This is especially useful when you know the remote procedure call you will make will take a while to return. Under these circumstances, you send it off, get busy doing other stuff in the meantime (making other calls, fetching records from a database, drinking tea, and so on), and then wait for the results when there's absolutely nothing else left to do. Of course, you can run such calls on your own node if you need to:

```
(cthulu@ferdmbp)5> MinTime = rpc:async_call(node(), timer, sleep, [30000]).
<0.48.0>
(cthulu@ferdmbp)6> lists:sort([a,c,b]).
[a,b,c]
```

```
(cthulu@ferdmbp)7> rpc:yield(MinTime).
... <long wait> ...
ok
```

If by any chance you want to use the `yield/1` function with a timeout value, call `rpc:nb_yield(Key, Timeout)` instead. To poll for results, use `rpc:nb_yield(Key)` (which is equivalent to `rpc:nb_yield(Key,0)`):

```
(cthulu@ferdmbp)8> Key2 = rpc:async_call(node(), timer, sleep, [30000]).
<0.52.0>
(cthulu@ferdmbp)9> rpc:nb_yield(Key2).
timeout
(cthulu@ferdmbp)10> rpc:nb_yield(Key2).
timeout
(cthulu@ferdmbp)11> rpc:nb_yield(Key2).
timeout
(cthulu@ferdmbp)12> rpc:nb_yield(Key2, 1000).
timeout
(cthulu@ferdmbp)13> rpc:nb_yield(Key2, 100000).
... <long wait> ...
{value,ok}
```

If you don't care about the result, you can use `rpc:cast(Node, Mod, Fun, Args)` to send a command to another node and forget about it.

But what if what we want is to call more than one node at a time? Let's add three nodes to our little cluster: `minion1`, `minion2`, and `minion3`. Those are `cthulu`'s minions. When we want to ask them questions, we need to send three different calls, and when we want to give orders, we must cast three times. That's pretty bad, and it doesn't scale with very large armies.

The trick is to use two `rpc` functions for calls and casts, respectively `rpc:multicall(Nodes, Mod, Fun, Args)` (with an optional `Timeout` argument) and `rpc:eval_everywhere(Nodes, Mod, Fun, Args)`:

```
(cthulu@ferdmbp)14> nodes().
[lovecraft@ferdmbp, minion1@ferdmbp, minion2@ferdmbp, minion3@ferdmbp]
(cthulu@ferdmbp)15> rpc:multicall(nodes(), erlang, is_alive, []).
{[true,true,true,true],[]}
```

This tells us that all four nodes are alive (and none were unavailable for an answer). The left side of the tuple is alive, the right side isn't. Yeah, `erlang:is_alive()` just returns whether the node it runs on is alive or not, which might look a bit weird. Yet again, remember that in a distributed setting, `alive` means "can be reached," not "is running." Then let's say `cthulu` isn't really appreciative of its minions and decides to kill them, or rather, talk them into killing themselves. This is an order, and so it's cast.

For this reason, we use `eval_everywhere/4` with a call to `init:stop()` on the minion nodes:

```
(cthulu@ferdmbp)16> rpc:eval_everywhere([minion1@ferdmbp, minion2@ferdmbp,
minion3@ferdmbp], init, stop, []).
abcast
(cthulu@ferdmbp)17> rpc:multicall([lovecraft@ferdmbp, minion1@ferdmbp,
minion2@ferdmbp, minion3@ferdmbp], erlang, is_alive, []).
{[true],[minion1@ferdmbp, minion2@ferdmbp, minion3@ferdmbp]}
```

When we ask again for who is alive, only one node remains: the `lovecraft` node. The minions were obedient creatures.

The `rpc` module has a few more interesting functions, but the core uses were covered here. If you want to know more, I suggest that you comb through the documentation for the module.

Burying the Distribunomicon

That's it for most of the basics on distributed Erlang. There are a lot of things to think about and a lot of attributes to keep in mind. Whenever you need to develop a distributed application, ask yourself which of the distributed computing fallacies you could potentially run into (if any). If a customer asks you to build a system that handles netsplits while staying consistent *and* available, you know that you need to either calmly explain the CAP theorem or run away (possibly by jumping through a window, for a maximal effect).

Generally, applications where a thousand isolated nodes can do their job without communicating or depending on each other will provide the best scalability. The more inter-node dependencies created, the harder it becomes to scale the system, no matter what kind of distribution layer you have. This is just like zombies (no, really!). Zombies are terrifying because of how many there are and how difficult it can be to kill them as a group. Even though individual zombies can be very slow and far from menacing, a horde can do considerable damage, even if it loses many of its zombie members. Groups of human survivors can do great things by combining their intelligence and communicating together, but each loss they suffer is more taxing on the group and its ability to survive.

That being said, you have the tools required to get going. Chapter 27 introduces distributed OTP applications. This kind of application provides a takeover and failover mechanism for hardware failures, but not general distribution. It's a lot more like respawning your dead zombie than anything else.

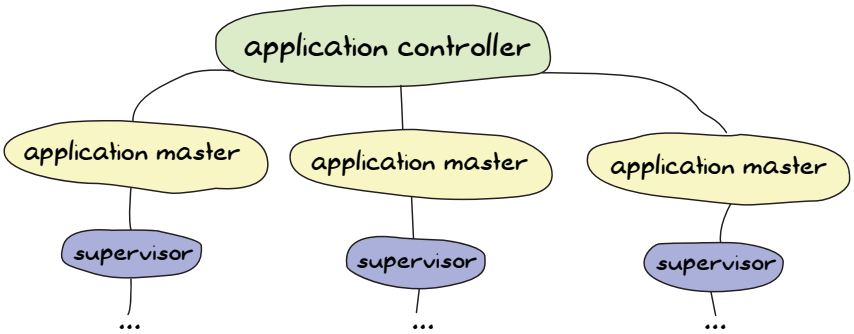
27

DISTRIBUTED OTP APPLICATIONS

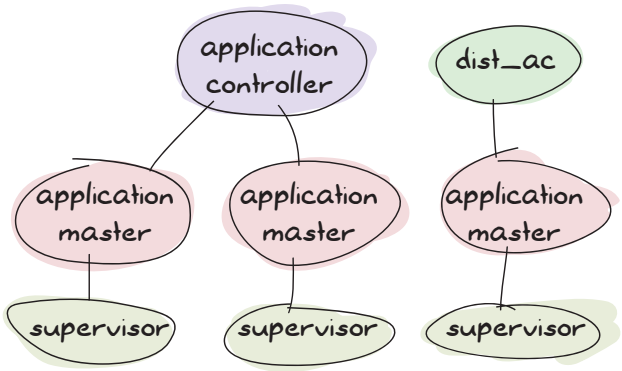
Although Erlang leaves us with a lot of work to do to build a distributed system, it still provides a few solutions. One of these is the concept of *distributed OTP applications*. Distributed OTP applications, or just *distributed applications* in the context of OTP, allow us to define takeover and failover mechanisms. In this chapter, we'll cover what that means and how it works, and write a little app to demonstrate these concepts.

Adding More to OTP

In Chapter 19, we briefly discussed the structure of an application as something using a central application controller dispatching to application masters, each monitoring a top-level supervisor for an application, like this:



In standard OTP applications, the application can be loaded, started, stopped, or unloaded. In distributed applications, we change how things work. Now the application controller shares its work with the *distributed application controller*, another process sitting next to it (usually called `dist_ac`), as shown here:



Depending on the application file, the ownership of the application will change. A `dist_ac` will be started on all nodes, and all `dist_ac` processes will communicate with each other. What they talk about is not too relevant, except for one thing. With standard applications, the four application statuses are being loaded, started, stopped, and unloaded. Distributed applications split the idea of a started application into *started* and *running*. The difference between the two is that you could define an application to be global within a cluster. An application of this kind can run on only one node at a time, while regular OTP applications don't care about what's happening on other nodes. As such, a distributed application will be started on all nodes of a cluster, but running on only one.

What does this mean for the nodes where the application is started without being run? The only thing they do is wait for the node of the running application to die. This means that when the node that runs the app dies, another node starts running it instead. This approach can avoid interruption of services by moving around different subsystems.

Taking and Failing Over

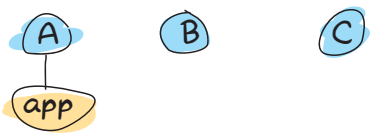
There are two important concepts handled by distributed applications: *failover* and *takeover*.

A failover is the idea of restarting an application somewhere other than where it stopped running. This is a particularly valid strategy when you have redundant hardware. You run something on a main computer or server, and if it fails, you move it to a backup one. In larger-scale deployments, you might instead have 50 servers running your software (all at maybe 60 to 70 percent load), and expect the running ones to absorb the load of the failing ones. The concept of failing over is mostly important in the former case, and somewhat less interesting in the latter one.

The second important concept of distributed OTP applications is the takeover. Taking over is the act of a dead node coming back from the dead, being known to be more important than the backup nodes (maybe it has better hardware), and deciding to run the application again. This is usually done by gracefully terminating the backup application and starting the main one instead.

NOTE *In terms of distributed programming fallacies, distributed OTP applications assume that when there is a failure, it is likely due to a hardware failure, and not a netsplit. If you deem netsplits more likely than other hardware failures, you must be aware of the possibility that the application is running both as a backup and main one, and that funny things could happen when the network issue is resolved. Maybe distributed OTP applications aren't the right mechanism for you in these cases.*

Let's imagine that we have a system with three nodes, where only the first one is running a given application:



The nodes `B` and `C` are declared to be backup nodes in case `A` dies, which we pretend just happened:

For a brief moment, nothing is running. After a while, `B` realizes this and decides to take over the application:

That's a failover. Then, if `B` dies, the application gets restarted on `C`:

Another failover, and all is well.

Now suppose that `A` comes back up. `C` is running the app happily, but `A` is the node we defined to be the main one. This is when a takeover occurs. The app is willingly shut down on `C` and restarted on `A`:

And so on for all other failures.

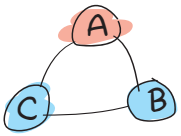
One obvious issue is how terminating applications all the time like this is likely to result in losing important state. Sadly, that's your problem. You'll need to think of places where to put and share all that vital state before things break down. The OTP mechanism for distributed applications makes no special case for that.

With these concepts in mind, let's move on to making things work in practice.

The Magic 8 Ball

A Magic 8 Ball is a simple toy that you shake to get divine and helpful answers. You ask questions like "Will my favorite sports team win the game tonight?" and the ball you shake replies something like "Without a doubt." You can then safely bet your house's value on the final score. Other questions like "Should I make careful investments for the future?" could return "That is unlikely" or "I'm not sure." The Magic 8 Ball has been vital in the Western world's political decision making in the past few decades, and it is only natural that we use it as an example for fault tolerance.

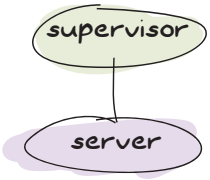
Our implementation won't make use of real-life switching mechanisms for automatically finding servers, such as round-robin DNS servers or load balancers. Instead, we'll stay within pure Erlang and have three nodes (denoted as `A`, `B`, and `C`) in a distributed OTP application. The `A` node will represent the main node running the Magic 8 Ball server, and the `B` and `C` nodes will be the backup nodes:



Whenever `A` fails, the Magic 8 Ball application should be restarted on either `B` or `C`, and both nodes will still be able to use it transparently.

Building the Application

Before setting up things for distributed OTP applications, we'll build the application itself. It's going to be mind-bogglingly naive in its design:



In total, we'll have three modules: the supervisor, the server, and the application callback module to start things.

The Supervisor Module

The supervisor will be rather trivial. We'll call it `m8ball_sup` (as in *Magic 8 Ball Supervisor*), and we'll put it in the *src/* directory of a standard OTP application:

```
-module(m8ball_sup).
-behaviour(supervisor).
-export([start_link/0, init/1]).

start_link() ->
    supervisor:start_link({global,?MODULE}, ?MODULE, []).

init([]) ->
    {ok, {{one_for_one, 1, 10},
          [{m8ball,
            {m8ball_server, start_link, []},
            permanent,
            5000,
```

```
            worker,
            [m8ball_server]
          }]}}.
```

This is a supervisor that will start a single server (`m8ball_server`), a permanent worker process. It's allowed one failure every 10 seconds.

The Server Module

The Magic 8 Ball server will be a bit more complex. We'll build it as a `gen_server` with the following interface:

```
-module(m8ball_server).
-behaviour(gen_server).
-export([start_link/0, stop/0, ask/1]).
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         code_change/3, terminate/2]).

%%%%%%%%%%%%%%%%
%%% INTERFACE %%%
%%%%%%%%%%%%%%%%
start_link() ->
    gen_server:start_link({global, ?MODULE}, ?MODULE, [], []).

stop() ->
    gen_server:call({global, ?MODULE}, stop).

ask(_Question) -> % The question doesn't matter!
    gen_server:call({global, ?MODULE}, question).
```

Notice how the server is started using `{global, ?MODULE}` as a name and how it's accessed with the same tuple for each call. That's the `global` module we saw in Chapter 26, applied to behaviors.

Next come the callbacks, the real implementation. The Magic 8 Ball should randomly pick one of many possible replies from some configuration file. We want a configuration file because it should be easy to add or remove answers as we wish.

First, if we want to do things randomly, we'll need to set up some randomness as part of our `init` function:

```
%%%%%%%%%%%%%%%%
%%% CALLBACKS %%%
%%%%%%%%%%%%%%%%
init([]) ->
    <<A:32, B:32, C:32>> = crypto:rand_bytes(12),
    random:seed(A,B,C),
    {ok, []}.
```

We used this pattern before in Chapter 23. Here, we're using 12 random bytes to set up the initial random seed to be used with the `random:uniform/1` function.

The next step is to read the answers from the configuration file and pick one. As discussed in Chapter 19, the easiest way to set up some configuration is through the app file (in the `env` tuple). Here's how we're going to do this:

```
handle_call(question, _From, State) ->
    {ok, Answers} = application:get_env(m8ball, answers),
    Answer = element(random:uniform(tuple_size(Answers)), Answers),
    {reply, Answer, State};
handle_call(stop, _From, State) ->
    {stop, normal, ok, State};
handle_call(_Call, _From, State) ->
    {noreply, State}.
```

The first clause shows what we want to do. We expect to have a tuple with all the possible answers within the `answers` value of the `env` tuple. Why a tuple? Simply because accessing elements of a tuple is a constant time operation, while obtaining them from a list is linear (and thus takes longer on larger lists). We then send back the answer.

NOTE *The server reads the answers with `application:get_env(m8ball, answers)` on each question asked. If you were to set new answers with a call like `application:set_env(m8ball, answers, {"yes","no","maybe"})`, the three answers would instantly be the possible choices for future calls. Reading them once at startup should be somewhat more efficient in the long run, but it will mean that the only way to update the possible answers is to restart the application or add a special call to do it.*

You should have noticed by now that we don't actually care about the question asked; it's not even passed to the server. Because we're returning random answers, it is entirely useless to copy the question from process to process. We're just saving work by ignoring it entirely. We still leave the answer there because it will make the final interface feel more natural. We could also trick our Magic 8 Ball into always returning the same answer for the same question if we felt like it, but we won't bother with that for this example.

The rest of the module is pretty much the same as usual for a generic `gen_server` doing nothing:

```
handle_cast(_Cast, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.
```

Now we can get to the more serious stuff, namely the application file and the callback module. We'll begin with the latter, *m8ball.erl*:

```
-module(m8ball).
-behaviour(application).
-export([start/2, stop/1]).
-export([ask/1]).

%%%%%%%%%%%%%%%%%
%%% CALLBACKS %%%
%%%%%%%%%%%%%%%%%
start(normal, []) ->
    m8ball_sup:start_link().

stop(_State) ->
    ok.

%%%%%%%%%%%%%%%%%
%%% INTERFACE %%%
%%%%%%%%%%%%%%%%%
ask(Question) ->
    m8ball_server:ask(Question).
```

That was easy. Here's the associated app file, *m8ball.app*:

```
{application, m8ball,
 [{vsn, "1.0.0"},
  {description, "Answer vital questions"},
  {modules, [m8ball, m8ball_sup, m8ball_server]},
  {applications, [stdlib, kernel, crypto]},
  {registered, [m8ball, m8ball_sup, m8ball_server]},
  {mod, {m8ball, []}},
  {env, [
    {answers, {<<"Yes">>, <<"No">>, <<"Doubtful">>,
               <<"I don't like your tone">>, <<"Of course">>,
               <<"Of course not">>, <<"*backs away slowly and runs away*">>}}
  ]}
 ]}.
```

We depend on `stdlib` and `kernel`, like all OTP applications, and also on `crypto` for our random seeds in the server. Notice how the answers are all in a tuple that matches the tuples required in the server. In this case, the answers are all binaries, but the string format doesn't really matter—a list would work as well.

Making the Application Distributed

So far, everything has been like building a perfectly normal OTP application. We have very few changes to make to our files to turn the normal

application into a distributed OTP application. In fact, we have only one function clause to add, back in the *m8ball.erl* module:

```
%%%%%%%%%%%%%%%%
%%% CALLBACKS %%%
%%%%%%%%%%%%%%%%

start(normal, []) ->
    m8ball_sup:start_link();
start({takeover, _OtherNode}, []) ->
    m8ball_sup:start_link().
```

The `{takeover, OtherNode}` argument is passed to `start/2` when a more important node takes over a backup node. In the case of the Magic 8 Ball app, it doesn't really change anything, and we can just start the supervisor all the same.

Recompile your code, and it's pretty much ready.

But hold on, how do we define which nodes are the main ones and which ones are backups? The answer is in configuration files. Because we want a system with three nodes (`a@yourhost`, `b@yourhost`, and `c@yourhost`), we'll need three configuration files (name them *a.config*, *b.config*, and *c.config*, and then put them all in *config/* inside the application directory):

```
[{kernel,
  [{distributed, [{m8ball,
                   5000,
                   [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]},
   {sync_nodes_mandatory, [b@ferdmbp, c@ferdmbp]},
   {sync_nodes_timeout, 30000}
  ]}].
```

```
[{kernel,
  [{distributed, [{m8ball,
                   5000,
                   [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]},
   {sync_nodes_mandatory, [a@ferdmbp, c@ferdmbp]},
   {sync_nodes_timeout, 30000}
  ]}].
```

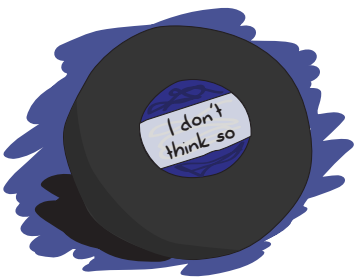
```
[{kernel,
  [{distributed, [{m8ball,
                   5000,
                   [a@ferdmbp, {b@ferdmbp, c@ferdmbp}]}]},
   {sync_nodes_mandatory, [a@ferdmbp, b@ferdmbp]},
   {sync_nodes_timeout, 30000}
  ]}].
```

The general structure is always the same:

```
[{kernel,
  [{distributed, [{AppName,
                   TimeOutBeforeRestart,
                   NodeList}]},
   {sync_nodes_mandatory, NecessaryNodes},
   {sync_nodes_optional, OptionalNodes},
   {sync_nodes_timeout, MaxTime}
  ]}].
```

The `NodeList` value can usually take a form like `[A, B, C, D]` for `A` to be the main one, `B` as the first backup, `C` as the next one, and `D` as the last. Another syntax is possible, giving a list of like `[A, {B, C}, D]`, so `A` is still the main node, `B` and `C` are equal secondary backups, and then comes the rest.

The `sync_nodes_mandatory` tuple will work in conjunction with `sync_nodes_timeout`. When you start a distributed VM with values set for this, the VM will stay locked up until all the mandatory nodes are also up and locked. Then they will be synchronized and things will start going. If it takes more than `MaxTime` to get all the nodes up, then they will all crash before starting.



There are a lot more options available, and I recommend looking into the kernel application documentation (*http://www.erlang.org/doc/man/kernel_app.html*) if you want to know more about them.

Let's try running the `m8ball` application. If you're not sure 30 seconds is enough to boot all three VMs, you can increase `sync_nodes_timeout` as you see fit. Then start three VMs:

```
$ erl -sname a -config config/a -pa ebin/
```

```
$ erl -sname b -config config/b -pa ebin/
```

```
$ erl -sname c -config config/c -pa ebin/
```

As you start the third VM, the other VMs should all unlock at once. Go into each of the three VMs, and turn by turn, start both `crypto` and `m8ball` with `application:start(AppName)`.

Now you should be able to call the Magic 8 Ball from any of the connected nodes:

```
(a@ferdmbp)3> m8ball:ask("If I crash, will I have a second life?").
<<"I don't like your tone">>
```

```
(a@ferdmbp)4> m8ball:ask("If I crash, will I have a second life, please?").
<<"Of Course">>
```

```
(c@ferdmbp)3> m8ball:ask("Am I ever gonna be good at Erlang?").
<<"Doubtful">>
```

How motivational. To see how things are with our applications, call `application:which_applications()` on all nodes. Only node `a` should be running it:

```
(b@ferdmbp)3> application:which_applications().
[{crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS  CXC 138 10","1.18.1"},
 {kernel,"ERTS  CXC 138 10","2.15.1"}]
```

```
(a@ferdmbp)5> application:which_applications().
[{m8ball,"Answer vital questions","1.0.0"},
 {crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS  CXC 138 10","1.18.1"},
 {kernel,"ERTS  CXC 138 10","2.15.1"}]
```

The `c` node should show the same thing as the `b` node in this case. Now if you kill the `a` node (just ungracefully close the window that holds the Erlang shell), the application should no longer be running there. Let's see where it is instead:

```
(c@ferdmbp)4> application:which_applications().
[{crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS  CXC 138 10","1.18.1"},
 {kernel,"ERTS  CXC 138 10","2.15.1"}]
(c@ferdmbp)5> m8ball:ask("where are you?!").
<<"I don't like your tone">>
```

That's expected, as `b` is higher in the priorities. After 5 seconds (we set the timeout to 5000 milliseconds), `b` should be showing the application as running:

```
(b@ferdmbp)4> application:which_applications().
[{m8ball,"Answer vital questions","1.0.0"},
 {crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS  CXC 138 10","1.18.1"},
 {kernel,"ERTS  CXC 138 10","2.15.1"}]
```

It still runs fine. Now kill `b` in the same barbaric manner that you used to get rid of `a`, and `c` should be running the application after 5 seconds:

```
(c@ferdmbp)6> application:which_applications().
[{m8ball,"Answer vital questions","1.0.0"},
```

```
 {crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS  CXC 138 10","1.18.1"},
 {kernel,"ERTS  CXC 138 10","2.15.1"}]
```

If you restart node `a` with the same command we used before, it will hang. The configuration file specifies we need `b` back for `a` to work. If you can't expect nodes to all be up that way, you will need to make either `b` or `c` optional. So if we start both `a` and `b`, the application should automatically come back, right?

```
(a@ferdmbp)4> application:which_applications().
[{crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS  CXC 138 10","1.18.1"},
 {kernel,"ERTS  CXC 138 10","2.15.1"}]
(a@ferdmbp)5> m8ball:ask("is the app gonna move here?").
<<"Of course not">>
```

Aw, shucks. The thing is, for the mechanism to work, the application needs to be started *as part of the boot procedure of the node*. You could, for instance, start node `a` that way for things to work:

```
$ erl -sname a -config config/a -pa ebin -eval 'application:start(crypto), application:start(m8ball)'
... <snip> ...
(a@ferdmbp)1> application:which_applications().
[{m8ball,"Answer vital questions","1.0.0"},
 {crypto,"CRYPTO version 2","2.1"},
 {stdlib,"ERTS  CXC 138 10","1.18.1"},
 {kernel,"ERTS  CXC 138 10","2.15.1"}]
```

Here's how it looks from node `c`'s side:

```
=INFO REPORT==== 8-Jan-2013::19:24:27 ===
    application: m8ball
    exited: stopped
    type: temporary
```

That's because the `-eval` option is evaluated as part of the boot procedure of the VM. Obviously, a cleaner way to do it would be to use releases to set things up right, but the example would be pretty cumbersome if it had to combine everything we have seen before.

Just remember that, in general, distributed OTP applications work best when working with releases that ensure that all the relevant parts of the system are in place.

As I mentioned earlier, in the case of many applications (the Magic 8 Ball included), it's sometimes simpler to just have many instances running at once and synchronizing data, rather than forcing an application to run in only a single place. It's also simpler to scale the system once that design has been picked. If you need some failover/takeover mechanism, distributed OTP applications might be just what you need.

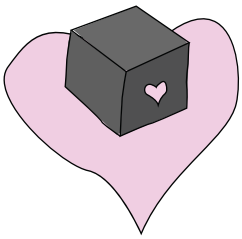
28

COMMON TEST FOR UNCOMMON TESTS

Back in Chapter 24, we explored how to use EUnit to do unit and module testing, and even some concurrent testing. At that point, EUnit started to show its limits. Complex setups and longer tests that needed to interact with each other became problematic. Plus, EUnit does not provide any help for handling distributed Erlang and all of its power. Fortunately, another test framework exists, and it's more appropriate for the heavy lifting we now want to do.

What Is Common Test?

As programmers, we enjoy treating our programs as black boxes. Many of us would define the core principle behind a good abstraction as being able to replace whatever we've written with an anonymous black box. You put something in the box; you get something out of it. You don't care how it works on the inside, as long as you get what you want.



In the testing world, this has an important connection to how we like to test systems. When we were working with EUnit, we used the approach of treating a module as a black box, where we tested only the exported functions and none of the ones inside, which are not exported. I've also given examples of testing items as a white box, as in the case of the Process Quest player module's tests, where we could look at the innards of the module to make its testing simpler. This was necessary because the interaction of all the moving parts inside the box made testing it from the outside very complex.

That was for modules and functions. What if we zoom out a bit? Let's fiddle with our scope in order to see the broader picture. What if we want to test a library? What if it's an application? Even broader, what if it's a complete system? Then we need a tool that is more adept at doing something called *system testing*.

EUnit is a pretty good tool for white box testing at a module level. It's a decent tool to test libraries and OTP applications. It's possible to use it for system testing and black box testing, but it's not optimal.

Common Test is pretty damn good at system testing. It's decent for testing libraries and OTP applications, and it's possible, but not optimal, to use it to test individual modules. So the smaller the size of what you test, the more appropriate (and flexible and fun) EUnit will be. The larger your test is, the more appropriate (and flexible and, uh, somewhat fun) Common Test will be.

You might have heard of Common Test before and tried to understand it from the documentation provided with Erlang/OTP. Then you likely gave up soon after. Don't worry. The problem is that Common Test is very powerful and has an accordingly long user guide. At the time of this writing, most of its documentation appears to be coming from internal documentation from the days when it was used only within the walls of Ericsson. In fact, its documentation is more of a reference manual for people who already understand it than a tutorial.

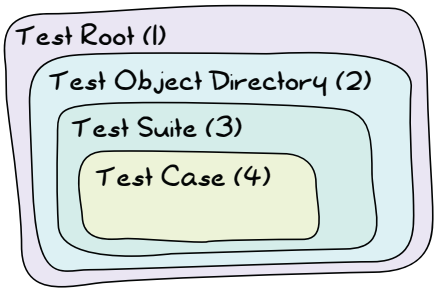
In order to properly learn Common Test, we'll start from the simplest parts of it and slowly grow our way to system tests. But before we get started, let's take a brief look at how Common Test is organized.

Common Test Structure

Because Common Test is appropriate for system testing, it will assume two things:

- We will need data to instantiate our stuff.
- We will need a place to store all that "side effect-y" stuff we do because we're messy people.

Because of these assumptions, Common Test will usually be organized as follows:

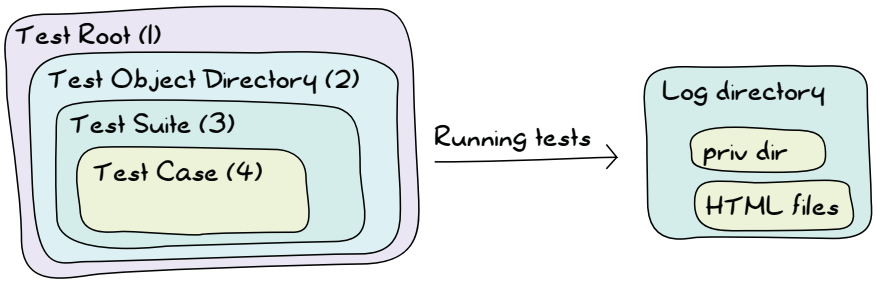


The test case is the simplest part. It's a bit of code that either fails or succeeds. If the case crashes, the test is unsuccessful (how surprising). Otherwise, the test case is considered successful.

In Common Test, test cases are single functions. All these functions live in a test suite (3), which is a module that takes care of regrouping related test cases together. Each test suite will then live in a directory—the test object directory (2). The test root (1) is a directory that contains many test object directories, but due to the nature of OTP applications often being developed individually, many Erlang programmers tend to omit that layer.

Let's go back to our two assumptions: we need to instantiate stuff, and then mess up stuff. Each test suite is a module that ends with `_SUITE`. If we were to test the Magic 8 Ball application from Chapter 27, we might call the suite `m8ball_SUITE`. Each suite is allowed to have a data directory, usually named *<Module>_SUITE_data/*. In the case of the Magic 8 Ball app, the data directory would have been *m8ball_SUITE_data/*. That directory contains anything you want that may be useful to the tests.

What about the side effects? Well, because we might run tests many times, Common Test develops its structure a bit more:



Whenever you run the tests, Common Test will find some place to log stuff (usually the current directory, but we'll cover how to configure that in "Test Specifications" on page 501). For this purpose, it will create a unique directory where you can store your data. That directory (*priv dir* in the structure diagram), along with the data directory, will be passed as part of

some initial state to each of your tests. You're then free to write whatever you want in that private directory, and then inspect it later, without running the risk of overwriting something important or the results of former test runs.

Enough with this architectural material; we're ready to write our first test suite.

Creating a Simple Test Suite

We'll begin with a simple test suite with two test cases. Create a directory named *ct/* (or whatever you like—this is a free country, after all). That directory will be our test root. Inside it, make a directory named *demo/* for the simpler tests we'll use as examples. This will be our test object directory.

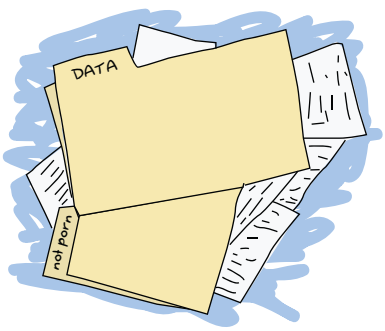
Inside the test object directory, we'll begin with a module named *basic_SUITE.erl*, to see the most basic stuff doable in Common Test. You can omit creating the *basic_SUITE_data/* directory—we won't need it for this run, and Common Test won't complain.

Here's what the module looks like:

```
-module(basic_SUITE).
-include_lib("common_test/include/ct.hrl").
-export([all/0]).
-export([test1/1, test2/1]).

all() -> [test1, test2].

test1(_Config) ->
    1 = 1.

test2(_Config) ->
    A = 0,
    1/A.
```

Let's study it step by step. First, we need to include the file `"common_test/include/ct.hrl"`. That file contains a few useful macros, and even though `basic_SUITE` doesn't use them, it's usually a good habit to include that file.

Then we have the function `all/0`. That function returns a list of test cases. It's basically what tells Common Test, "Hey, I want to run these test cases!" EUnit would do it based on the name (`*_test()` or `*_test_()`). Common Test does it with an explicit function call.



What about these `_Config` variables? They're unused for now, but for your own personal knowledge, they contain the initial state your test cases will require. That

state is literally a proplist, and it initially contains two values: `data_dir` and `priv_dir`—the directory we have for our static data and the one where we can mess around.

Running the Tests

We can run the tests either from the command line or from an Erlang shell. From the command line, call `ct_run -suite Name_SUITE`.

NOTE *In Erlang/OTP versions before R15 (released around December 2011), the default command was `run_test` instead of `ct_run` (although some systems had both already). The change was made to help minimize the risk of name clashes with other applications by moving to a slightly less generic name.*

By running this command, we get the following:

```
$ ct_run -suite basic_SUITE
... <snip> ...
Common Test: Running make in test directories...
Recompile: basic_SUITE
... <snip> ...
Testing ct.demo.basic_SUITE: Starting test, 2 test cases

- - - - - - - - - - - - - - - - - - - - - - - - -
basic_SUITE:test2 failed on line 13
Reason: badarith
- - - - - - - - - - - - - - - - - - - - - - - - -

Testing ct.demo.basic_SUITE: *** FAILED *** test case 2 of 2
Testing ct.demo.basic_SUITE: TEST COMPLETE, 1 ok, 1 failed of 2 test cases

Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/all_runs.html... done
```

And we find that one of our two test cases fails. We also see that we apparently inherited a bunch of HTML files. Before looking into what this is about, let's see how to run the tests from the Erlang shell:

```
$ erl
... <snip> ...
1> ct:run_test([{suite, basic_SUITE}]).
... <snip> ...
Testing ct.demo.basic_SUITE: Starting test, 2 test cases

- - - - - - - - - - - - - - - - - - - - - - - - -
basic_SUITE:test2 failed on line 13
Reason: badarith
- - - - - - - - - - - - - - - - - - - - - - - - -
... <snip> ...
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/demo/all_runs.html... done
ok
```

I've removed a bit of the output, but the shell gives exactly the same result as the command-line version.

Let's see what's going on with these HTML files:

```
$ ls
all_runs.html
basic_SUITE.beam
basic_SUITE.erl
ct_default.css
ct_run.NodeName.YYYY-MM-DD_20.01.25/
ct_run.NodeName.YYYY-MM-DD_20.05.17/
index.html
variables-NodeName
```

Oh what the hell did Common Test do to our beautiful directory? It is a shameful thing to look at. We have two directories there. Feel free to explore them if you feel adventurous, but cowards like me will prefer to instead look at either the *all_runs.html* or the *index.html* file. The former links to indexes of all iterations of the tests you ran, and the latter links to the newest runs only. Pick one, and then click around in a browser (or press around if you don't believe in a mouse as an input device) until you find the test suite with its two tests:

Num	Module	Case	Log	Time	Result	Comment
1	basic_SUITE	test1	< >	0.000s	Ok	
2	basic_SUITE	test2	< >	0.000s	FAILED	{basic_SUITE,test2,13} badarith
	TOTAL			0.139s	**FAILED**	1 Ok, 1 Failed of 2

You see that `test2` failed. If you click the underlined line number, you'll see a raw copy of the module. If you click the `test2` link, you'll see a detailed log of what happened:

```
=== source code for basic_SUITE:test2/1
=== Test case started with:
basic_SUITE:test2(ConfigOpts)
=== Current directory is "Somewhere on my computer"
=== Started at 2013-01-20 20:05:17
[Test Related Output]
=== Ended at 2013-01-20 20:05:17
=== location [{basic_SUITE,test2,13},
              {test_server,ts_tc,1635},
              {test_server,run_test_case_eval1,1182},
              {test_server,run_test_case_eval,1123}]
=== reason = bad argument in an arithmetic expression
  in function  basic_SUITE:test2/1 (basic_SUITE.erl, line 13)
  in call from test_server:ts_tc/3 (test_server.erl, line 1635)
  in call from test_server:run_test_case_eval1/6 (test_server.erl, line 1182)
  in call from test_server:run_test_case_eval/9 (test_server.erl, line 1123)
```

The log lets you know precisely what failed, and it is much more detailed than what we got in the Erlang shell. So, if you're a shell user, you'll find Common Test extremely painful to use. If you're a person more prone to using GUIs, then it will be more fun for you.

But enough wandering around pretty HTML files, let's see how to test with some more state.

NOTE *If you ever feel like traveling back in time without the help of a time machine, download a version of Erlang prior to R15B and use Common Test with it. You'll be astonished to see that your browser and the logs' style brings you back to the late 1990s.*

Testing with State

As explained in Chapter 24, EUnit has these things called *fixtures*, where we give a test case some special instantiation (setup) and teardown code to be called before and after the case, respectively.

Common Test follows that concept. However, instead of having EUnit-style fixtures, it relies on two functions:

- The setup function, `init_per_testcase/2`
- The teardown function, `end_per_testcase/2`

To see how they are used, create a new test suite called `state_SUITE` (still under the *demo/* directory), and add the following code:

```
-module(state_SUITE).
-include_lib("common_test/include/ct.hrl").

-export([all/0, init_per_testcase/2, end_per_testcase/2]).
-export([ets_tests/1]).

all() -> [ets_tests].

init_per_testcase(ets_tests, Config) ->
    TabId = ets:new(account, [ordered_set, public]),
    ets:insert(TabId, {andy, 2131}),
    ets:insert(TabId, {david, 12}),
    ets:insert(TabId, {steve, 12943752}),
    [{table,TabId} | Config].

end_per_testcase(ets_tests, Config) ->
    ets:delete(?config(table, Config)).

ets_tests(Config) ->
    TabId = ?config(table, Config),
    [{david, 12}] = ets:lookup(TabId, david),
    steve = ets:last(TabId),
    true = ets:insert(TabId, {zachary, 99}),
    zachary = ets:last(TabId).
```

This is a little normal ETS test checking a few `ordered_set` concepts. What's interesting about it is the two new functions: `init_per_testcase/2` and `end_per_testcase/2`. Both functions need to be exported in order to be called. If they are exported, the functions will be called for *all* test cases in a module. You can separate them based on the arguments. The first one is the name of the test case (as an atom), and the second one is the `Config` proplist that you can modify.

NOTE *To read from `Config`, rather than using `proplists:get_value/2`, the Common Test include file has a `?config(Key, List)` macro that returns the value matching the given key. The macro is a wrapper around `proplists:get_value/2` and is documented as such, so you know you can deal with `Config` as a proplist without worrying about it ever breaking.*

As an example, if we had tests `a`, `b`, and `c` and wanted a setup and teardown function for only the first two tests, our `init` function might look like this:

```
init_per_testcase(a, Config) ->
    [{some_key, 124} | Config];
init_per_testcase(b, Config) ->
    [{other_key, duck} | Config];
init_per_testcase(_, Config) ->
    %% Ignore for all other cases.
    Config.
```

And we would handle the `end_per_testcase/2` function similarly.

Looking back at `state_SUITE`, you can see the test case, but what's interesting is how we instantiate the ETS table. We don't specify an heir, and yet the tests run without a problem after the `init` function is finished.

As discussed in Chapter 25, ETS tables are usually owned by the process that started them. In this case, we leave the table as it is. If you run the tests, you'll see the suite succeeds.

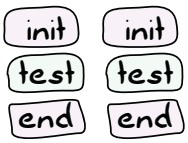
What we can infer from this is that the `init_per_testcase` and `end_per_testcase` functions run in the same process as the test case itself. You can thus safely do things like set links or start tables within these functions without worrying about having your things breaking the way they would if they were running in different processes. What about errors in the test case? Fortunately, crashing in your test case won't stop Common Test from cleaning up and calling the `end_per_testcase` function, with the exception of `kill` exit signals.

So far, our work with Common Test is at least equal to, if not more than, what we can do with EUnit, at least in terms of flexibility. Although we don't get all the nice assertion macros, we have fancier reports, similar fixtures, and that private directory where we can write stuff from scratch. What more do we want?

NOTE *If you end up feeling like outputting information to help you debug or just show progress in your tests, you'll quickly find out that* `io:format/1-2` *prints only to the HTML logs, not to the Erlang shell. If you want to do both (with free timestamps included), use the function* `ct:pal/1-2`*. It works like* `io:format/1-2`*, but prints to both the shell and logs.*

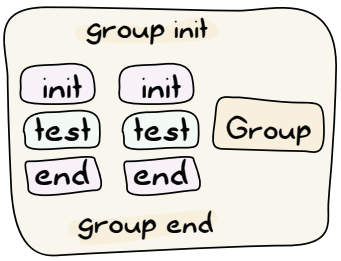
Test Groups

Right now, our test structure within a suite might look like this:



What if we have many test cases with similar needs in terms of some `init` functions, but some different parts in them? Well, the easy way to do it is to copy/paste and modify, but this will be a real pain to maintain. Moreover, what if we want to run many tests in parallel or in random order instead of one after the other? There's no easy way to do that based on what we've seen so far. This is pretty much the same kind of problem that could limit our use of EUnit.

To solve these issues, we have *test groups*. Common Test test groups allow us to regroup some tests hierarchically. Even more, they can regroup some groups within other groups, as shown here:



In this hierarchy, a group has its own initialization and termination functions, wrapping many tests or other groups. This allows us to define some common environment to a bunch of related tests, or even groups of other tests. For example, if half of your tests need to connect to a database with a given configuration, and the other half with another configuration, setting this configuration with different groups would be the most efficient way to do it.

Defining Test Groups

For this approach to work, we need test groups. First, we add a `groups()` function to declare all of the groups:

```
groups() -> ListOfGroups.
```

Here's what `ListOfGroups` should be:

```
[{GroupName, GroupProperties, GroupMembers}]
```

And in more detail, here's what this could look like:

```
[{test_case_street_gang,
  [],
  [simple_case, more_complex_case]}].
```

That's a tiny test case street gang. Here's a more complex one:

```
[{test_case_street_gang,
  [shuffle, sequence],
  [simple_case, more_complex_case,
   emotionally_complex_case,
   {group, name_of_another_test_group}]}].
```

This one specifies two properties: `shuffle` and `sequence`. We'll look at what they mean soon.

The example also shows a group including another group. This assumes that the `groups()` function might be a bit like this:

```
groups() ->
    [{test_case_street_gang,
      [shuffle, sequence],
      [simple_case, more_complex_case, emotionally_complex_case,
       {group, name_of_another_test_group}]},
     {name_of_another_test_group,
      [],
      [case1, case2, case3]}].
```

You can also define the group inline within another group:

```
[{test_case_street_gang,
  [shuffle, sequence],
  [simple_case, more_complex_case,
   emotionally_complex_case,
   {name_of_another_test_group,
    [],
    [case1, case2, case3]}
  ]}].
```

That's getting a bit complex, right? Read the examples carefully, and it should get simpler with time. Keep in mind that nested groups are not mandatory, and you can avoid them if you find them confusing.

But wait, how do you use such groups? By putting them in the `all/0` function:

```
all() -> [some_case, {group, test_case_street_gang}, other_case].
```

This way, Common Test will be able to know whether it needs to run a single test case or a group of them.

Test Group Properties

The preceding examples used some test group properties, including `shuffle`, `sequence`, and an empty list. The following group properties are available:

`empty list / no option`
: The test cases in the group are run one after the other. If a test fails, the others after it in the list are run.

`shuffle`
: This runs the test in a random order. The random seed (the initialization value) used for the sequence will be printed in the HTML logs, in the form `{A,B,C}`. If a particular sequence of tests fails and you want to reproduce it, use that seed in the HTML logs and change the `shuffle` option to be `{shuffle, {A,B,C}}`. That way, you can reproduce random runs in their precise order if you ever need to do so.

`parallel`
: The tests are run in different processes. Be careful, because if you forget to export the `init_per_group` and `end_per_group` functions, Common Test will silently ignore this option.

`sequence`
: Using this option doesn't necessarily mean that the tests are run in order, but rather that if a test fails in the group's list, then all the other subsequent tests are skipped. This option can be combined with `shuffle` if you want any random test failing to stop the ones that follow.

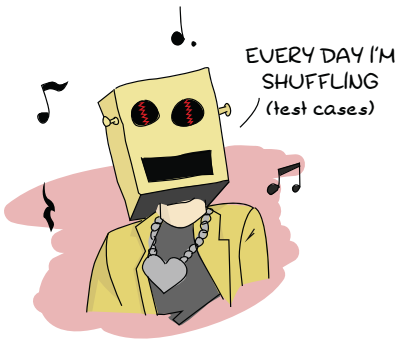
`{repeat, Times}`
: This repeats the group `Times` times. You could run the whole test case sequence in parallel nine times by using the group properties `[parallel, {repeat, 9}]`. `Times` can also have the value `forever`, although "forever" is a bit of a lie, as it can't defeat concepts such as hardware failure or heat death of the universe (ahem).

{repeat_until_any_fail, N}
: This runs all the tests until one of them fails or they have been run `N` times. `N` can also be `forever`.

{repeat_until_all_fail, N}
: This works the same as the preceding option, but the tests may run until all cases fail.

{repeat_until_any_succeed, N}
: This is also the same as the preceding options, except the tests may run until at least one case succeeds.

{repeat_until_all_succeed, N}
: I think you can guess what this one does, but just in case: It's the same as the preceding options, except that the test cases may run until they all succeed.



Honestly, that's quite a bit of content for test groups, and I feel an example would be appropriate here.

The Meeting Room

To use test groups, we'll create a meeting room-booking module:

```
-module(meeting).
-export([rent_projector/1, use_chairs/1, book_room/1,
         get_all_bookings/0, start/0, stop/0]).
-record(bookings, {projector, room, chairs}).

start() ->
    Pid = spawn(fun() -> loop(#bookings{}) end),
    register(?MODULE, Pid).

stop() -> ?MODULE ! stop.

rent_projector(Group) -> ?MODULE ! {projector, Group}.

book_room(Group) -> ?MODULE ! {room, Group}.

use_chairs(Group) -> ?MODULE ! {chairs, Group}.
```

These basic functions will call a central registry process. They will allow us to book the room, rent a projector, and put dibs on chairs. For the sake of the exercise, we're in a large organization with one hell of a corporate structure. Because of this, three different people are responsible for

the projector, the room, and the chairs, but there is one central registry. Because of this structure, you can't book all items at once, but must do it by sending three different messages.

To know who booked what, we can send a message to the registry in order to get all the values:

```
get_all_bookings() ->
    Ref = make_ref(),
    ?MODULE ! {self(), Ref, get_bookings},
    receive
        {Ref, Reply} ->
            Reply
    end.
```

The registry itself looks like this:

```
loop(B = #bookings{}) ->
    receive
        stop -> ok;
        {From, Ref, get_bookings} ->
            From ! {Ref, [{room, B#bookings.room},
                          {chairs, B#bookings.chairs},
                          {projector, B#bookings.projector}]},
            loop(B);
        {room, Group} -> loop(B#bookings{room=Group});
        {chairs, Group} -> loop(B#bookings{chairs=Group});
        {projector, Group} -> loop(B#bookings{projector=Group})
    end.
```

And that's it.

To book everything for a successful meeting, we would need to successively call the functions:

```
1> c(meeting).
{ok,meeting}
2> meeting:start().
true
3> meeting:book_room(erlang_group).
{room,erlang_group}
4> meeting:rent_projector(erlang_group).
{projector,erlang_group}
5> meeting:use_chairs(erlang_group).
{chairs,erlang_group}
6> meeting:get_all_bookings().
[{room,erlang_group},
 {chairs,erlang_group},
 {projector,erlang_group}]
```

This doesn't seem right, though. You possibly have this lingering feeling that things could go wrong. In many cases, if we make the three calls fast enough, we should obtain everything we want without a problem. If two

people do it at once and there are short pauses between the calls, it seems possible that two (or more) groups might try to rent the same equipment at the same time.

Oh no! Suddenly, the programmers might end up having the projector, while the board of directors has the room, and the human resources department managed to rent all of the chairs. All resources are tied up, but no one can do anything useful!

We won't worry about fixing that problem. Instead, we'll work on trying to demonstrate that it's present with a Common Test suite.

The suite, named *meeting_SUITE.erl*, will be based on the simple idea of trying to provoke a race condition that will mess up the registration. We'll have three test cases, each representing a group. Carla will represent women, Mark will represent men, and a dog will represent a group of animals that somehow decided it wanted to hold a meeting with human-made tools:

```
-module(meeting_SUITE).
-include_lib("common_test/include/ct.hrl").
 ...
carla(_Config) ->
    meeting:book_room(women),
    timer:sleep(10),
    meeting:rent_projector(women),
    timer:sleep(10),
    meeting:use_chairs(women).

mark(_Config) ->
    meeting:rent_projector(men),
    timer:sleep(10),
    meeting:use_chairs(men),
    timer:sleep(10),
    meeting:book_room(men).

dog(_Config) ->
    meeting:rent_projector(animals),
    timer:sleep(10),
    meeting:use_chairs(animals),
    timer:sleep(10),
    meeting:book_room(animals).
```

We don't care whether these tests actually test something. They are just there to use the `meeting` module (which we'll put in place for the tests soon) and try to generate wrong reservations.

To find out if we have a race condition between all of these tests, we'll use the `meeting:get_all_bookings()` function in a fourth and final test:

```
all_same_owner(_Config) ->
    [{_, Owner}, {_, Owner}, {_, Owner}] = meeting:get_all_bookings().
```

This one does pattern matching on the owners of all different objects that can be booked, trying to see whether they are actually booked by the same owner. This is a desirable thing if we are looking for efficient meetings.

How do we move from having four test cases in a file to something that works? We'll need to make clever use of test groups. First, because we want a race condition, we know we'll need to have a bunch of tests running in parallel. Second, given we have a requirement to see the problem from the race condition, we'll need to either run `all_same_owner` many times during the whole debacle or only after it, to look with despair at the aftermath.

I chose the latter:

```
all() -> [{group, clients}, all_same_owner].

groups() -> [{clients, [parallel, {repeat, 10}], [carla, mark, dog]}].
```

This creates a `clients` group of tests, with the individual tests being `carla`, `mark`, and `dog`. They're going to run in parallel, 10 times each.

You see that we include the group in the `all/0` function, and then put `all_same_owner`. That's because, by default, Common Test will run the tests and groups in `all/0` in the order they were declared.

But wait! We forgot to start and stop the meeting process itself. To do that, we'll need to have a way to keep a process alive for all tests, regardless of whether it's in the `clients` group. The solution is to nest things one level deeper, in another group:

```
all() -> [{group, session}].

groups() -> [{session, [], [{group, clients}, all_same_owner]},
             {clients, [parallel, {repeat, 10}], [carla, mark, dog]}].

init_per_group(session, Config) ->
    meeting:start(),
    Config;
init_per_group(_, Config) ->
    Config.

end_per_group(session, _Config) ->
    meeting:stop();
end_per_group(_, _Config) ->
    ok.
```

We use the `init_per_group` and `end_per_group` functions to specify that the `session` group (which now runs `{group, clients}` and `all_same_owner`) will work with an active meeting. Don't forget to export the two setup and teardown functions; otherwise, nothing will run in parallel.

All right, let's run the tests and see what we get:

```
1> ct_run:run_test([{suite, meeting_SUITE}]).
... <snip> ...
Common Test: Running make in test directories...
... <snip> ...
TEST INFO: 1 test(s), 1 suite(s)
Testing ct.meeting.meeting_SUITE: Starting test (with repeated test cases)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
meeting_SUITE:all_same_owner failed on line 50
Reason: {badmatch,[{room,men},{chairs,women},{projector,women}]}
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Testing ct.meeting.meeting_SUITE: *** FAILED *** test case 31
Testing ct.meeting.meeting_SUITE: TEST COMPLETE, 30 ok, 1 failed of 31 test cases
... <snip> ...
ok
```

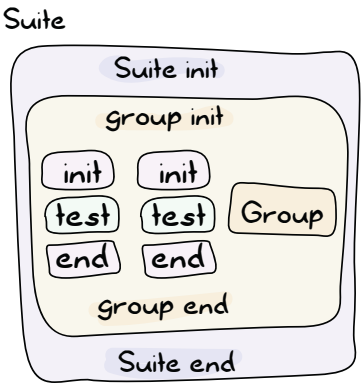
Good! We have a `badmatch` with three tuples with different items owned by different people. Moreover, the output tells us it's the `all_same_owner` test that failed. I think that's a pretty good sign that `all_same_owner` crashed as planned.

If you look at the HTML log, you'll be able to see all the runs with the exact test that failed and for what reason. Click the test name, and you'll get the right test run.

NOTE *An important thing to know about test groups is that while the `init` functions of test cases run in the same process as the test case, the `init` functions of groups run in processes distinct from the tests. This means that whenever you initialize actors that get linked to the process that spawned them, you must make sure to first unlink them. In the case of ETS tables, you need to define an heir to make sure it doesn't disappear. This applies to all other items that are attached to a process, such as sockets and file descriptors.*

Test Suites Redux

Can we use our test suites in a way that is better than nesting groups and manipulating how we run things in terms of hierarchy? Not really, but even so, we'll add another level with the test suite itself:



We have two additional functions: `init_per_suite(Config)` and `end_per_suite(Config)`. These, like all the other `init` and `end` functions, aim to give more control over initialization of data and processes.

The `init_per_suite/1` and `end_per_suite/1` functions will run only once, respectively, before and after all of the groups or test cases. They will be useful when dealing with general state and dependencies that will be required for all tests. This can include manually starting applications you depend on, for example.

Test Specifications

There's something you might have found pretty annoying if you looked at your test directory after running tests: a ton of files scattered around the directory for your logs—CSS files, HTML logs, directories, test run histories, and so on. It would be pretty neat to have a nice way to store these files in a single directory.

Another issue is that so far we've run tests from a test suite. We have not seen a good way to do it with many test suites at once, or even ways to run only one or two cases or groups from a suite (or from many suites).

Of course, if I'm bringing up these issues, it's because I have a solution for them. There are ways to handle them from both the command line and the Erlang shell, and you can find them in the documentation for `ct_run` (*http://www.erlang.org/doc/man/ct_run.html*). However, instead of going into ways to manually specify everything for each time you run the tests, we'll employ *test specifications*.

Test specifications are special files that let you detail everything about how you want to have the tests run, and they work with the Erlang shell and the command line. The test specification can be put in a file with any extension you want (although I personally fancy *.spec* files).

Specification File Contents

The spec files will contain Erlang tuples, much like a consult file (a file containing Erlang terms, which can be parsed by using `file:consult/1`). Here are some of the items a spec file can contain:

`{include, IncludeDirectories}`
: When Common Test automatically compiles suites, this option lets you specify where it should look for include files in order to make sure they are there. The `IncludeDirectories` value must be a string (list) or a list of strings (list of lists).

`{logdir, LoggingDirectory}`
: When logging, all logs should be moved to the `LoggingDirectory`, a string. Note that the directory must exist before the tests are run; otherwise, Common Test will complain.

{suites, Directory, Suites}
: This finds the given suites in `Directory`. Suites can be an atom (`some_SUITE`), a list of atoms, or the atom `all` to run all the suites in a directory.

{skip_suites, Directory, Suites, Comment}
: This subtracts a list of suites from those previously declared and skips them. The `Comment` argument is a string explaining why you decided to skip them. This comment will be put in the final HTML logs. The tables will show, in yellow, "SKIPPED: *Reason*," where *Reason* is whatever `Comment` contained.

{groups, Directory, Suite, Groups}
: This is an option to pick only a few groups from a given suite. The `Groups` variable can be a single atom (the group name) or `all` for all groups. The value can also be more complex, letting you override the group definitions inside `groups()` within the test case by picking a value like `{GroupName, [parallel]}`, which will override `GroupName`'s options for `parallel`, without needing to recompile tests.

{groups, Directory, Suite, Groups, {cases,Cases}}
: This option is similar to the previous one, but it lets you specify some test cases to include in the tests with `Cases`, which can be a single case name (an atom), a list of names, or the atom `all`.

{skip_groups, Directory, Suite, Groups, Comment}
: This command was added in R15B and documented in R15B01. It allows you to skip test groups, much like the `skip_suites` for suites.

{skip_groups, Directory, Suite, Groups, {cases,Cases}, Comment}
: This is similar to the previous option, but with specific test cases to skip on top of it. It also has been available only since R15B.

{cases, Directory, Suite, Cases}
: This runs specific test cases from a given suite. `Cases` can be an atom, a list of atoms, or `all`.

{skip_cases, Directory, Suite, Cases, Comment}
: This is similar to `skip_suites`, except you choose specific test cases to avoid with this one.

{alias, Alias, Directory}
: Because it gets very annoying to write all these directory names (especially if they're full names), Common Test lets you replace them with aliases (atoms). This is pretty useful way to be concise.

Creating a Spec File

Let's try a simple example. First, add a *ct/logs/* directory, on the same level as *ct/demo/*. Unsurprisingly, that's where our Common Test logs will be moved.

Here's a possible test specification for all our tests so far, saved under the imaginative name *spec.spec*:

```
{alias, demo, "./demo/"}.
{alias, meeting, "./meeting/"}.
{logdir, "./logs/"}.

{suites, meeting, all}.
{suites, demo, all}.
{skip_cases, demo, basic_SUITE, test2, "This test fails on purpose"}.
```

This spec file declares two aliases: `demo` and `meeting`, which point to the two test directories we have. We put the logs inside *ct/logs/*, our newest directory. Then we ask to run all suites in the *meeting* directory, which, coincidentally, is the `meeting_SUITE` suite.

Next on the list are the two suites inside the *demo* directory. We ask to skip `test2` from the `basic_SUITE` suite, given it contains a division by zero that we know will fail.

Running Tests with a Spec File

To run the tests, you can use `ct_run -spec spec.spec` (or `run_test` for versions of Erlang before R15) from the command line, or you can use the function `ct:run_test([{spec, "spec.spec"}])`. from the Erlang shell:

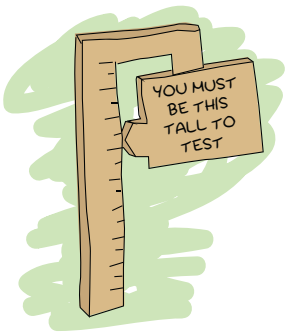
```
Common Test: Running make in test directories...
... <snip> ...
TEST INFO: 2 test(s), 3 suite(s)

Testing ct.meeting: Starting test (with repeated test cases)

- - - - - - - - - - - - - - - - - - - - - - - - - -
meeting_SUITE:all_same_owner failed on line 51
Reason: {badmatch,[{room,men},{chairs,women},{projector,women}]}
- - - - - - - - - - - - - - - - - - - - - - - - - -

Testing ct.meeting: *** FAILED *** test case 31
Testing ct.meeting: TEST COMPLETE, 30 ok, 1 failed of 31 test cases

Testing ct.demo: Starting test, 3 test cases
Testing ct.demo: TEST COMPLETE, 2 ok, 0 failed, 1 skipped of 3 test cases

Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/all_runs.html... done
```

If you take the time to look at the logs, you'll see two directories for the different test runs. One of them will have a failure—that's the

meeting that failed as expected. The other one will have one success and one skipped case, of the form *1 (1/0)*. Generally, the format is *TotalSkipped (IntentionallySkipped/SkippedDueToError)*. In this case, the skip happened from the spec file, so it goes on the left. If it happened because one of the many `init` functions failed, then it would be on the right.

Common Test is starting to look like a pretty decent testing framework, but it would be nice to be able to use our distributed programming knowledge and apply it.

Large-Scale Testing



Common Test supports having distributed tests. Before going hog wild and writing a bunch of code, let's see what's offered. Well, there isn't *that* much. The gist of it is that Common Test lets you start tests on many different nodes, but also has ways to dynamically start these nodes and have them watch each other.

As such, the distributed features of Common Test are really useful when you have large test suites that should be run in parallel on many nodes. This is often worth the effort to save time or because the code will run in production environments that are on different computers—automated tests that reflect this are desired.

When tests go distributed, Common Test requires the presence of a central node (the *CT master*) in charge of all the other nodes. Everything will be directed from there, from starting nodes, ordering tests to be run, gathering logs, and so on.

The first step to get things going that way is to expand our test specifications so they become distributed. We're going to add a couple new tuples: `{node, NodeAlias, NodeName}` and `{init, NodeAlias, Options}`.

`{node, NodeAlias, NodeName}` is much like `{alias, AliasAtom, Directory}` for test suites, groups, and cases, except it's used for node names. Both `NodeAlias` and `NodeName` need to be atoms. This tuple is especially useful when `NodeName` needs to be a long node name, since it would be quite annoying to have it duplicated in its entire form dozens of times over a given spec file.

`{init, NodeAlias, Options}` is a more complex tuple. This is the option that lets you start nodes. `NodeAlias` can be a single node alias or a list of many of them. The `Options` are those available to the `ct_slave` module.

Here are a few of the options available:

`{username, UserName}` and `{password, Password}`
: Using the host part of the node given by `NodeAlias`, Common Test will try to connect to the given host over SSH (on port 22) using the username and password, and run from there.

{startup_functions, [{M,F,A}]}
: This option defines a list of functions to be called as soon as the other node has booted.

{erl_flags, String}
: This sets standard flags that you would want to pass to the `erl` application when you start it. For example, if you wanted to start a node with `erl -env ERL_LIBS ../ -config conf_file`, the option would be `{erl_flags, "-env ERL_LIBS ../ -config config_file"}`.

{monitor_master, true | false}
: If the CT master stops running and this option is set to `true`, the slave node will also be taken down. I recommend using this option if you're spawning the remote nodes; otherwise, they will keep running in the background if the master dies. Moreover, if you run tests again, Common Test will be able to connect to these nodes, and there will be some state attached to them.

{boot_timeout, Seconds}, {init_timeout, Seconds}, and {startup_timeout, Seconds}
: These three options let you wait for different parts of the starting of a remote node. The boot timeout is about how long it takes before the node becomes pingable, with a default value of 3 seconds. The init timeout is an internal timer where the new remote node calls back the CT master to say that it's up. By default, it lasts 1 second. Finally, the startup timeout tells Common Test how long to wait for the functions you defined earlier as part of the `startup_functions` tuple.

{kill_if_fail, true | false}
: This option will react to one of the three preceding timeouts. If any of them are triggered, Common Test will abort the connection, skip tests, and so on, but not necessarily kill the node, unless the option is set to `true`. Fortunately, that's the default value.

NOTE *All these options are provided by the `ct_slave` module. It is possible to define your own module to start slave nodes, as long as it respects the right interface.*

So, we have a lot of options for remote nodes, which contributes to giving Common Test its distributed power. You're able to boot nodes with about as much control as what you would get doing it by hand in the shell. Still, there are more options for distributed tests, although they're not for booting nodes:

```
{include, Nodes, IncludeDirs}
{logdir, Nodes, LogDir}
{suites, Nodes, DirectoryOrAlias, Suites}
{groups, Nodes, DirectoryOrAlias, Suite, Groups}
{groups, Nodes, DirectoryOrAlias, Suite, GroupSpec, {cases,Cases}}
{cases, Nodes, DirectoryOrAlias, Suite, Cases}
```

```
{skip_suites, Nodes, DirectoryOrAlias, Suites, Comment}
{skip_cases, Nodes, DirectoryOrAlias, Suite, Cases, Comment}
```

These are similar to the options we've already seen, except that they can optionally take a node argument to add more detail. That way, you can decide to run some suites on a given node, others on different nodes, and so on. This could be useful when doing system testing with different nodes running different environments or parts of the system (such as databases or external applications).

Creating a Distributed Spec File

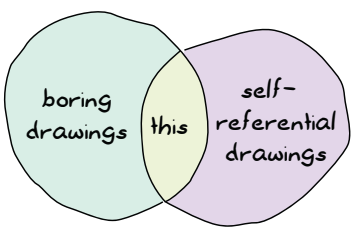
As a simple way to see how distributed testing works, let's turn the previous *spec.spec* file into a distributed one. Copy it as *dist.spec*, and then change it to look like this:

```
{node, a, 'a@ferdmbp.local'}.
{node, b, 'b@ferdmbp.local'}.

{init, [a,b], [{node_start, [{monitor_master, true}]}]}.

{alias, demo, "./demo/"}.
{alias, meeting, "./meeting/"}.

{logdir, [all_nodes, master], "./logs/"}.

{suites, [b], meeting, all}.
{suites, [a], demo, all}.
{skip_cases, [a], demo, basic_SUITE, test2, "This test fails on purpose"}.
```

In this version, we define two slave nodes, `a` and `b`, that need to be started for the tests. They do nothing special but make sure to die if the master dies. The aliases for directories remain the same as they were.

The `logdir` values are interesting. We did not declare a node alias as `all_nodes` or `master`, but yet, here they are. The `all_nodes` alias stands for all non-master nodes for Common Test; `master` stands for the master node itself. To truly include all nodes, `[all_nodes, master]` is required. (There's no clear explanation as to why these names were picked.)

The reason we used these `logdir` values is that Common Test will generate logs (and directories) for each of the slave nodes, and it will also generate a master set of logs, referring to the slave ones. We don't want any of these in directories other than *logs/*. Note that the logs for the slave nodes will be stored on each of the slave nodes



individually. In that case, unless all nodes share the same filesystem, the HTML links in the master's logs won't work, and you'll need to access each of the nodes to get their respective logs.

Last of all are the `suites` and `skip_cases` entries. They're pretty much the same as the previous ones, but adapted for each node. This way, you can skip some entries only on given nodes (which you know might be missing libraries or dependencies), or maybe more intensive ones where the hardware isn't up to the task.

Running Distributed Tests

To run distributed tests, we must start a distributed node with `-name` and use `ct_master` to run the suites (there is no way to run such tests using `ct_run`.):

```
$ erl -name ct
Erlang R15B (erts-5.9) [source] [64-bit] [smp:4:4] [async-threads:0] [hipe] [kernel-poll:false]

Eshell V5.9  (abort with ^G)
(ct@ferdmbp.local)1> ct_master:run("dist.spec").
=== Master Logdir ===
/Users/ferd/code/self/learn-you-some-erlang/ct/logs
=== Master Logger process started ===
<0.46.0>
Node 'a@ferdmbp.local' started successfully with callback ct_slave
Node 'b@ferdmbp.local' started successfully with callback ct_slave
=== Cookie ===
'PMIYERCHJZNZGSRJPVRK'
=== Starting Tests ===
Tests starting on: ['b@ferdmbp.local','a@ferdmbp.local']
=== Test Info ===
Starting test(s) on 'b@ferdmbp.local'...
=== Test Info ===
Starting test(s) on 'a@ferdmbp.local'...
=== Test Info ===
Test(s) on node 'a@ferdmbp.local' finished.
=== Test Info ===
Test(s) on node 'b@ferdmbp.local' finished.
=== TEST RESULTS ===
a@ferdmbp.local_____________________________finished_ok
b@ferdmbp.local_____________________________finished_ok

=== Info ===
Updating log files
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/index.html... done
Updating /Users/ferd/code/self/learn-you-some-erlang/ct/logs/all_runs.html... done
Logs in /Users/ferd/code/self/learn-you-some-erlang/ct/logs refreshed!
=== Info ===
Refreshing logs in "/Users/ferd/code/self/learn-you-some-erlang/ct/logs"... ok
[{"dist.spec",ok}]
```

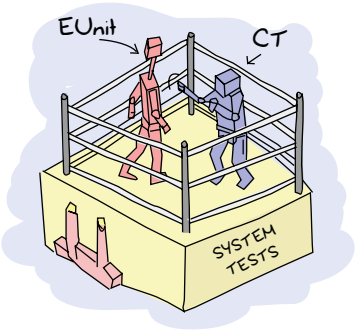
Note that Common Test will show all results as `ok` whether or not the tests actually succeeded. That is because `ct_master` shows only if it could contact all the nodes. The results themselves are actually stored on each individual node.

Also note that Common Test shows that it started nodes, and with what cookies it did so. If you try running tests again without first terminating the master, the following warnings are shown instead:

```
WARNING: Node 'a@ferdmbp.local' is alive but has node_start option
WARNING: Node 'b@ferdmbp.local' is alive but has node_start option
```

That's all right. It only means that Common Test is able to connect to remote nodes, but found no use for calling our `init` tuple from the test specification, given the nodes are already alive. There is no need for Common Test to actually start any remote nodes it will run tests on, but I usually find it useful to do so.

That's really the gist of distributed spec files. Of course, you can get into more complex cases, where you set up more complicated clusters and write amazing distributed tests. But as the tests become more complex, you can have less confidence in their ability to successfully demonstrate the properties of your software, simply because the tests themselves might contain more errors as they become more convoluted.



Integrating EUnit Within Common Test

Because sometimes EUnit is the best tool for the job, and sometimes Common Test is, it might be desirable for you to include one into the other.

While it's difficult to include Common Test suites within EUnit ones, the opposite is quite easy. The trick is that when you call `eunit:test(SomeModule)`, the function can return either `ok` when things work or `error` in case of any failure.

This means that to integrate EUnit tests into a Common Test suite, all you need to do is have a function a bit like this:

```
run_eunit(_Config) ->
    ok = eunit:test(TestsToRun).
```

And all your EUnit tests that can be found by the `TestsToRun` description will be run. If there's a failure, it will appear in your Common Test logs, and you'll be able to read the output to see what went wrong. It's that simple.

Is There More?

You bet there's more. Common Test is a very complex beast. There are ways to add configuration files for some variables; add hooks that run at many points during the test executions; use callbacks on events during the suites; and use modules to test over SSH (`ct_ssh`), telnet (`ct_telnet`), SNMP (`ct_snmp`), and FTP (`ct_ftp`).

This chapter only scratched the surface, but it is enough to get you started if you want to explore Common Test in more depth. A more complete document is the Common Test User's Guide, which comes with Erlang/OTP (*http://www.erlang.org/doc/apps/common_test/users_guide.html*). As I mentioned at the beginning of this chapter, the guide is hard to read on its own, but understanding the material covered in this chapter will help you figure out the documentation, without a doubt.