Learn You Some Erlang for Great Good!

A Beginner's Guide

Fred Hébert

San Francisco

DISTRIBUNOMICON

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.

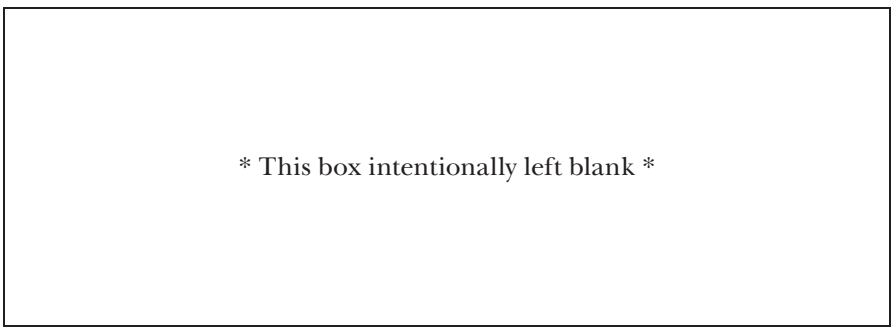
The problem is that a large message can hold back heartbeats. Too many large messages keeping heartbeats at bay for too long will eventually result in either of the nodes assuming the other is unresponsive and closing its connection to it. That's bad.

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.

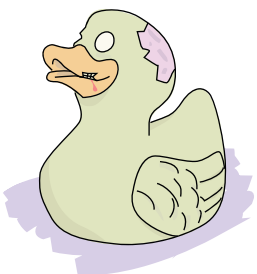
For Erlang, due to its initial use cases, no special care has been taken to do things like compress messages going cross-node (although the functions for it already exist). Instead, the original designers chose to let people implement their own communication layer if they required it. The responsibility 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.