

2010 Dr. Dobbs
Excellence Awards



NOMINATE YOURSELF TODAY!
YOU CAN'T WIN AN AWARD UNLESS YOU ENTER!

Dr. Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Using Erlang to Build Reliable, Fault Tolerant, Scalable Systems

A case study in rebuilding Yahoo! Harvester

By Arun Suresh and Jebu Ittiachen

October 12, 2009

URL: <http://www.ddj.com/high-performance-computing/220600332>

Jebu Ittiachen is a Principal Engineer at Yahoo!, working with the Content Platform Group. Jebu blogs at <http://blog.jebu.net> and tweets frequently from @jebui. Arun Suresh works for Yahoo! as a Technical Lead. He is currently developing next-generation platforms that power Yahoo! Properties. He tweets from @arun_suresh.

Harvester is a data acquisition component at Yahoo! that acquires data from various providers concurrently through a pull model from drop boxes via various protocols. Feeds can either be scheduled or fetched on demand.

While Harvester was originally written in Perl, Erlang's high-level concurrency constructs -- along with OTP design principles -- make it an ideal platform for building reliable, fault tolerant, and scalable applications like Harvester. The resulting service is more scalable, available, reliable, and able to comply with tighter service-level agreements (SLA) on a lighter code base and less expensive development efforts. In this article, we compare how the two systems perform side-by-side on a single box and showcase the higher ROI provided by the Erlang stack.

Harvester is modeled as a set of cooperating Perl processes. A Perl daemon (foreman) monitors a queue in the database for feeds that are scheduled to run. The foreman launches worker processes when a feed is ready for a fetch. A separate Perl daemon (conductor) schedules jobs into the queue. For scalability and high availability, all nodes run the foreman and conductor. To avoid concurrency problems, they use named locks in the database to avoid stepping on each other. Figure 1 presents a high-level architectural view of a queued batch processing system such as this.

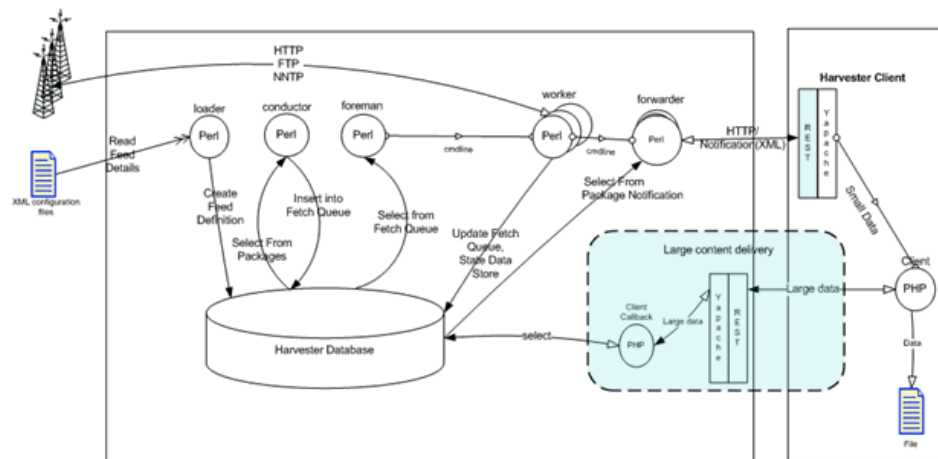


Figure1: Architectural view of the system.

The advantages of this approach include:

- Since each process is well isolated, a crash does not affect other processes.
- It scales horizontally reasonably well.
- There is no single point of failure (except the database).

Still, the current design has limited Harvester's adoption in areas such as mission-critical feeds with strict SLA. These drawbacks include:

- Perl processes are heavy, each taking almost three seconds to load.
- Resource usage per feed is high.
- Upgrades were extremely tricky given the nature of feeds running on the system.
- Load balancing across processes was skewed by proximity of write master.
- SLA misses under peak loads because of a combination of above-mentioned factors

Extracting More From the Current Implementation

To address these drawbacks, we took several steps, starting with tweaking the existing architecture with pooled Perl processes. We hoped this would reduce the process launch times by pooling worker processes that were launched, reusing it for the next run. However, the resource usage was still high per process. Considering that the processes are mostly I/O intensive, the huge number of spawned process meant neither the CPU nor the I/O was being effectively utilized and the OS was spending more time in process switching. Moreover, the database contention still remains. Bringing in a job management system into the architecture meant more maintenance overhead with no significant returns on resource utilization. Threads in Perl were explored for more efficient resource utilization but not advised by the gurus.

Stepping back we realized that the requirements were of a broader nature. We realized that what we needed was a:

- Programming model that simplified parallel processing.
- Well-tested distributed programming model for scaling horizontally.
- Reliable and guaranteed error handling mechanism.
- Framework that allowed code upgrades with no downtime.
- Efficient use of modern multicore commodity hardware.
- Easily integrate with the existing software stack at Yahoo!.

With this in mind, we considered a Java solution. The Java stack has a very good threading model, which has been proven in the field. However, mapping our problem to the Java application server model like Tomcat was not a natural fit because Harvester:

- Is not a web-app.
- Does a lot of network calls from within the container.
- Has a lot of long running calls.

Building it as a stand-alone threaded Java app would mean introducing the issues associated with the threaded programming model. The threading model is based on shared memory and locking to achieve consistency. Developing and deploying an error-free threaded program is tough, since hard-to-reproduce concurrency issues go hand-in-hand with shared memory. A message passing solution is not native to Java and we would need to bring in an external library to help us do messaging, in turn introducing a performance impact.

Also, current Java VM's are not well-suited for the new range of hardware sporting more cores and lower clock speeds. The code upgrade issue still remains and doing application upgrades involve taking a host down, upgrading it, and putting it back into the cluster.

Why Erlang?

Erlang is a concurrent programming language and its approach to concurrency is the actor model. It has very lightweight processes (not OS processes) and built-in message passing semantics among processes. Spawning a process takes a few microseconds and an Erlang VM can have thousands of these lightweight processes running simultaneously. Sending a message from one process to another is in the order of nano seconds. Each process is well isolated from others, and a rouge process does not bring the whole system down. Process can be externally managed and killed if it skews.

Furthermore, Erlang has a shared nothing programming model, which avoids locking issues. The processes are isolated and a process crash can be captured as a message and send to the parent process. The process-spawning model has built in support for distribution, an Erlang node can spawn a process on an authorized remote Erlang VM and send messages to it like a local process. Errors from remote nodes are also reported like normal local process failures.

Erlang is a soft real-time system, with a pre-emptive scheduler among the processes. Garbage collection (GC) is implemented in a simpler manner with cleanup happening at a process level. Since nothing is shared among the processes GC is cheaper as no object traversals are involved. This made it possible for us to meet strict SLA needs.

This process and messaging model enables us to run more feeds per box, drastically reducing the resource required per feed. Manageability and isolation at the process level allows us to reliably co-host multiple feeds.

Open Telecom Platform

The [Open Telecom Platform \(OTP\)](#) is a framework and set of principles for how to structure Erlang code in terms of processes, modules, and directories. A common pattern along which process are structured in an OTP application is the [Supervision tree](#). This is a model based on the idea of workers and supervisors.

Workers are encouraged to fail if it encounters a situation that it cannot handle. The architecture of Harvester is such that every Worker or Foreman spawned in Harvester is associated with a Supervisor. This ensures that if a worker ever crashes during a fetch, its supervisor can decide on restarting it. Since the restart functionality is already found in the standard library, no extra code was required to be written.

The OTP framework standardizes application patterns for concurrency-oriented applications. It defines patterns like `gen_server` for client server model, `gen_fsm` for finite state machines, `gen_event` for event handling, and so on. It also defines a release and packaging specifications for deploying and upgrading applications. The error-handling semantics are also abstracted in the OTP framework. Building Harvester following the OTP principles meant we get to reuse the well-tested framework for building and packaging Erlang applications.

Erlang comes from a telecom background where zero downtime is the norm. Consequently it has features for live code upgrade, which blends well with our objectives. Having almost zero downtime is essential to Harvester since it's a hosted platform with critical feeds from different properties and getting a downtime window is not easy. The release packaging and upgrade model of OTP gives us easy access to the live code upgrade features in Erlang.

There are other benefits as well. Erlang is a functional programming language that uses pattern matching for many tasks. It's used for extracting values from data structures, control-flow within functions, for receiving messages, and the like. The libraries that come with Erlang make extensive use of lists and higher order functions.

The Erlang prototype required that we write it in a purely functional style. Although the initial learning curve turned out to be steep, the effort reaped rich dividends. This allowed us to code in a more expressive and succinct manner.

To avoid database contention, the Erlang Harvester prototype assigns feeds to an Erlang node and the node is responsible for its lifecycle. If the node fails, the feed is reassigned to a different node by the monitoring component. The monitoring piece is lightweight as all it does is monitor status of the harvester nodes, and shuffles feeds to a node that is alive.

Since Erlang is a non-standard platform at Yahoo!, its interoperability with the existing software stack is a concern. Erlang has a well-defined model for interacting with external applications with Erlang ports. An Erlang port talks to an external program running in a separate OS process via STDIN and STDOUT. We integrated with internal libraries via a simple Perl wrapper communicating with an Erlang port.

Drawbacks

Erlang is not the perfect platform. It has its shares of weakness and most of it comes from the design goals behind the language. Erlang does not have a string data type, a string in Erlang is a list of integers. Distinguishing between a list of integers and a string is not possible. It provides a module for basic string manipulation, and regular expressions but they pale in comparison to Perl. The latest version of Erlang R13 has Unicode support and libraries to convert between various encodings.

Records are the Erlang equivalent of C structs, but it is implemented like an after-thought on the Erlang tuple data type which are static in nature. This does not fit well with the dynamic nature of Erlang run time. The Erlang syntax can take some getting used to, but when you've got it you can appreciate the power it hides.

Performance Comparisons

Table 1 presents the results of a comparison between the current Perl implementation and the Erlang prototype. Note that quad-core test machine was 100% loaded on all four cores when processing the above-mentioned loads. The prototype was more reliable with supervisor trees doing most of the error handling part (the errors are logged and workers die). The supervisor decides if an error is to be retried or flagged for operations to intervene.

<i>Feature</i>	<i>Existing implementation</i>	<i>Erlang prototype</i>
Concurrently feeds, scheduled every minute	40	1000+
Skew in fetch time when loaded	2-10 minutes	10sec
Fetch time	26sec	9sec
Max feeds per machine	40	1500

Table 1: Performance comparison

Conclusion

Erlang provides a set of programming constructs that fit well for a certain set of problems. Within Yahoo!, Erlang is being successfully used in BOSS, Delicious, MyBlogLog and FireEagle. Understanding the features and benefits of the Erlang model and having it on the list of tools available when designing your systems will help you to pick the right stack for your system.

References

- [Erlang for Concurrent Programming](#), Jim Larson, Google
- [The Erlang Reference Manual](#)
- [Making Reliable Distributed Systems In the Presence of Software Errors](#)
- A History of Erlang, by Joe Armstrong



Copyright © 2010 [United Business Media LLC](#)