Sendmail Meets Erlang: Experiences Using Erlang for Email Applications

Scott Lystig Fritchie, Jim Larson, and Nick Christenson: Sendmail, Inc.*
Debi Jones†
Lennart Öhman: Sjöland & Thyselius Telecom AB‡

October 3, 2000

Abstract

Our software engineering team needed to create a system that moves data from a set of legacy applications with diverse properties to data repositories scattered around the network. This system had to be highly concurrent, straightforward to extend, have high performance, and be coded rapidly by a small development staff. Because of these requirements, the authors embarked upon an experiment to write this application in Erlang. This paper describes what we did, why we did it, and what we learned over the course of our development effort. It is our hope that this chronicle may be useful to others thinking about coding in Erlang for the first time and to the incumbent Erlang community to hear an outsider's perspective on this fine language.

1 Introduction

In the fall of 1999, a development team from Sendmail, Inc. began developing a program that would act as an I/O request broker connecting a set of legacy applications to a set of distributed data repositories. Some of these legacy applications might operate under a UNIX `inetd`-like forking model, some might be multithreaded using one thread per connection, and some might be multithreaded using thread pools and event-driven programming techniques. The data passing through this program, which was creatively named the "Client Daemon", is multiplexed over several separate connections to remote data repositories. In essence, the Client Daemon acts as a "traffic cop" — marshaling, massaging, and redirecting data and data requests from a varied set of legacy applications to a set of distributed servers.

There were a significant number of additional constraints placed on this project:

- **Rapid Development.** This technology introduced some fairly radical notions of data movement and storage. The system needed to be demonstrated to work or to fail as soon as possible, to allow time for a redesign.
- **Early time to market.** Sendmail, Inc., like most software companies, is competing on "Internet time." We wanted to be able to get our system to market as quickly as possible.
- **Avoid proprietary, non-portable hardware and software.** Platforms such as VAXcluster [KLS86] or products from IBM, HP, and other vendors could achieve many of these goals using off-the-shelf proprietary solutions. However, this system must be straightforward to port to a variety of UNIX operating systems and perhaps even to Windows NT.

* `<scott@sendmail.com>`, `<npc@sendmail.com>`, and `<jim@sendmail.com>`, respectively.
† `<netbean@earthlink.net>`
‡ `<lennart.ohman@st.se>`

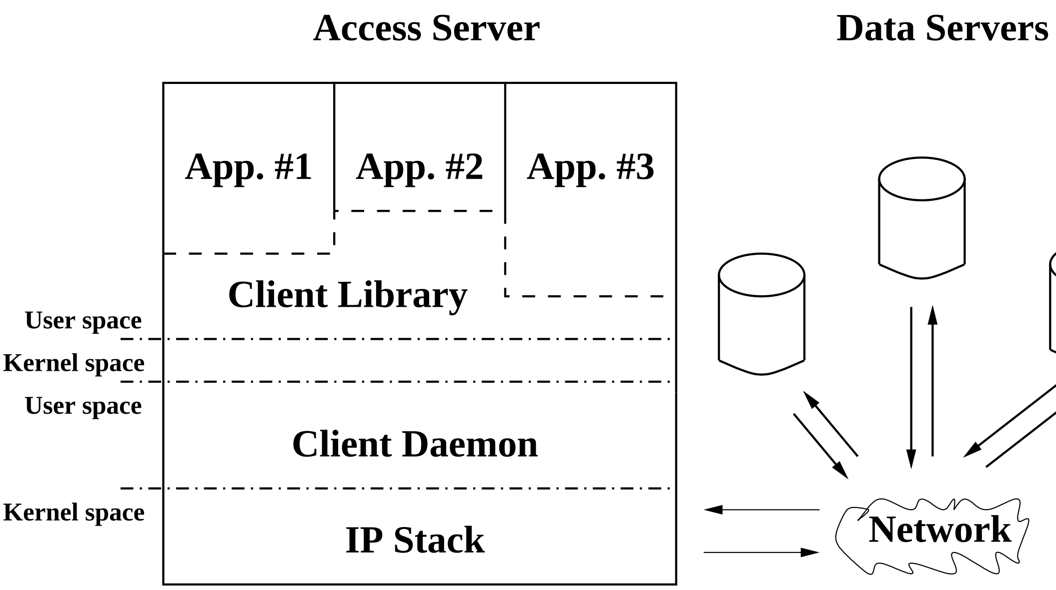

Figure 1: Process Architecture Sketch

- **High performance.** A distributed architecture will have higher overhead than a well-tuned architecture based on a single machine. Given that fundamental limitation, we want to make the overall system and each component as fast as possible. The Client Daemon initiates few data requests itself. The bulk of its work is taking data from one side (from the network or another application), massaging it, and passing it through to the other side. This means that its network interface and its IPC must be highly efficient.

- **Concurrency.** The Client Daemon will need to handle and track many simultaneous requests and responses from disparate sources. Each of these request/response pairs will take an unknown amount of time to complete.[1] In order to accommodate the various legacy applications, the Client Daemon will need to handle both synchronous and asynchronous communication modes, a notoriously difficult task in most programming languages.

- **Ease of Management.** There will be many nodes running a Client Daemon in a production environment. Therefore, we need some mechanism to tie all these systems together for monitoring and administration.

Figure 1 represents the overall architecture. The legacy applications are linked with a glue layer called the "Client Library". The Client Library uses IPC to move the data from these applications to the Client Daemon, and the Client Daemon moves the data over the network to remote Data Servers.

This paper describes the justification for and the procedures we went through to build the Client Daemon using Erlang. We chronicle our experiences, including what went well and what didn't. We provide some advice for other groups not terribly familiar with programming in Erlang and some suggestions for the Erlang community on how Erlang might be made more appealing to software development groups like ours.

[1] In fact, they may never complete. The Client Daemon must handle network timeouts in a context-sensitive manner.

2 Background

Since we and the company had developed many applications in C already, we had an intrinsic bias toward this language. Early on, we built a rudimentary prototype in C using the Libero [Lib] and SMT [SMT] tools by iMatix and using the familiar `rpcgen` and `libc`'s ONC RPC library to handle the data transport over the network to the remote Data Servers. The Libero/SMT finite-state machine approach seemed a viable way to manage the Client Daemon's growth if we decided we liked SMT, and we wouldn't lose much time if we didn't like it.

2.1 C Prototype

We confirmed our suspicions that the ONC RPC stub generator, `rpcgen`, generates some really ugly code. Furthermore, it assumes that all RPC calls are fully synchronous. Even modern `rpcgen` implementations capable of generating thread-safe stubs assume the calls are synchronous. As a result, we had to write our own RPC call mux/demux code, ignoring most of what `rpcgen` had created and bypassing much of the ONC RPC library's infrastructure.

After just over a month of coding, the Libero/SMT version of the Client Daemon was capable of interacting with both the applications and the Data Servers, although it was by no means ready for production use. At this point, we had learned a lot about RPC client stubs, RPC server stubs, XDR encoding, RPC fragment reassembly, and other RPC arcana, but the Client Daemon still had at least one obscure memory leak, its performance seemed slower than estimates predicted, and there were a great many core features that still needed to be implemented.

2.2 Erlang Discovered

Before starting the first prototype, we had conducted an exhaustive literature search to learn what lessons we could from the work of others. During this time, we came across references to the HTTP load balancer Eddie [Edd], which is written in Erlang. Erlang's built-in concurrency model was seductive: all the threading add-ons to C and C++ looked ugly by comparison. The "standard" Erlang libraries and OTP helped keep us from typing too much. Tony Rogvall gave us the source code to a full-featured ONC RPC library, allowing us to avoid a lot of XDR- and RPC-related drudge work.[2]

Also, around this time we had the opportunity to talk with a number of the software engineers at Bluetail A.B. who have written most of an email proxying system, called the Mail Robustifier [Blu], in Erlang. The fact that these folks had done significant work on an email-related project in Erlang increased our confidence in Erlang's viability. Their kindness to answer some general questions about coding such a project in Erlang helped us tremendously.

Two potential benefits of using Erlang were most appealing: higher programmer productivity and easier-to-maintain code. It was difficult to tell how many of the success stories from Ericsson and other companies using Erlang were worthwhile praise and how many were hype. Since our group was already familiar with a wide variety of arcane languages, we were quickly able to understand the reasons that these claims might be more than just smoke. We felt that if Erlang could live up to its promises, many of the goals of the project could be met much more easily with Erlang than with C, especially given our time-to-market concerns.

3 Implementation in Erlang

It's no surprise that software development managers are uncomfortable adopting new programming languages or techniques. Trying new and radical techniques with the industry's current time-to-market demands is usually a recipe for disaster. Sendmail, Inc.'s management was as skeptical as one would expect. Our dabbling with developing a second Client Daemon prototype in Erlang was greeted cautiously. Outside of our team,

[2] We have contributed our enhanced version of his package to the Erlang community. It should be available at `http://www.erlang.org/user.html` by the time of the conference.

nobody in our company had even heard of the language, much less knew anything about developing software in it.

3.1 The Second Prototype

The beginning of the learning curve was steep. The simple matter of writing a non-trivial program in a functional language is a radical change for people used to working with procedural languages such as C or Perl. We used parts of the OTP as best we understood them. The result was not pretty, but it worked.

Once we got over the startup costs of coming to terms with the new language, the Erlang prototype fell together quickly. After a month and a half, the Erlang Client Daemon had surpassed the Libero/SMT prototype in stability and feature set with comparable performance. We didn't discover any project-killing issues, so we recommended to our management that we complete our development of the Client Daemon in Erlang.

3.2 Convincing Management To Let Us Continue

Our team had the advantage of being small and closely-knit. The design and prototype development had been done in a "skunkworks"-like atmosphere: most of the company was preoccupied with other projects. Given the size of our team, the promise of writing a complex application using a relatively small number of lines of code, and our time-to-market constraints, our team agreed that the potential benefits were worth risking our project on Erlang.

Our management was skeptical about using Erlang for production code. First, they felt the discomfort normally associated with radical ideas. Second, no other developers knew it, which makes it more difficult to solicit advice or conduct code reviews in-house. Third, management felt that it might be hard to hire programmers for the team, since they would have to know both C and Erlang.

On the first and second points, we were still not entirely comfortable with the language ourselves, but both we and our management team were willing to set aside discomfort if the reasons for doing something new were compelling enough. On the issue of hiring, we pointed out that we had come up to speed fairly quickly. We felt that any programmer with an adventurous spirit could learn Erlang as quickly as we had (especially with our mentoring) and that our projections about time-to-market made this a worthwhile trade-off. Further, they were persuaded by our estimates about how quickly we could code and debug this application in Erlang as opposed to C. In the end, the rapid development schedule was able to compensate for the risk, since "If you're going to fail, fail early [Bro95]."

3.3 Further Development and Performance Tuning

We spent the next several months tightening up the code, improving Tony's RPC library, filling out features in the Client Daemon, improving the Client Library, and fixing bugs. While the core of the system was coming together, we had two outstanding concerns. The first was that we found the overall structure of the code to be vaguely unsettling, and the Client Daemon still didn't perform as well as we'd like.

Prior to starting work on this project, we had established performance goals. One of our really big concerns going into this project was that the Client Daemon would be doing a lot of data copying. There would be significant performance penalties if it could not do so efficiently. The default drivers in the Erlang emulator implement I/O via disk, pipes to spawned sub-processes, and TCP or UDP network connections. The only way to have direct communication with an unrelated UNIX process is through TCP or UDP sockets, which are not as efficient as other IPC mechanisms.

We created some crude programs that simulated how the Client Daemon might perform if it could interact with the outside world using UNIX domain sockets, through an `mmap()`-style DMA mechanism, and via other mechanisms. We saw some potential for improvement there, but we still didn't know what was consuming all of our systems' resources.

We used the Erlang profiler, `eprof`, to determine which processes were taking most of the CPU time. Unfortunately, it was difficult to measure CPU time (versus wall-clock time) consumed or a global context for the answer. We also had no insight as to how much time was spent in the runtime system for things like scheduling, memory management, linked-in drivers, and message-passing. To answer these questions, we compiled the virtual machine with `gprof` support. This gave us a global context for the performance data, but didn't give us any correlation with the Erlang code, only with the C code. For instance, we knew how much of the resources were taken up by garbage collection, but we did not know which processes or modules were producing the most garbage.

3.4 Outside Assistance

At this point, we felt that we could accelerate our progress with some outside help. Friends in the Erlang community recommended Lennart Öhman (one of the authors of this paper), an experienced Erlang developer and trainer.

After being briefed on what we were doing and why, Lennart first set out to explain current best practice regarding process hierarchy, including supervisor structure principles and process linking techniques.[3] Since the Client Daemon was not written using a strict top-down approach, but grew via more "organic" methods, we never looked at the entire process structure as a whole, and thus had a process hierarchy that was structured poorly. In hindsight, much of his advice seemed like common sense, and we probably would have figured it out eventually, but we never stumbled across it in our perusal of the current documentation and our inspection of other Erlang applications. Lennart's presentation was much more efficient than discovering the principles ourselves by trial and error.

We also received a great deal of training on parts of the OTP that we hadn't yet used. We learned quite a number of best practices for Erlang coding that we hadn't found documented anywhere or found only after we knew what to look for. Overall, this contributed a great deal to the organization of our code, making it more flexible, structured, and readable, and generally enabled us to think more clearly about its architecture.

The performance issues proved to be more difficult. We discussed several ideas on how to speed things up. One idea was to run the legacy applications as an Erlang I/O port under the Client Daemon, allowing communication with that process using a pipe rather than an IP socket. However, that technique won't work with those legacy applications that fork. We considered adding an Erlang driver that would allow zero-copy I/O of bulk data in and out of the virtual machine (in most cases), but the time to implement such a modification to the system ran against our time-to-market constraints and has been shelved for future consideration.

It was brought to our attention that there haven't been many Erlang projects that are both I/O intensive and have had externally-driven performance goals. Our application's workload seems atypical in today's Erlang usage, excepting Eddie and Bluetail's products. While it has been disappointing not to reach our *a priori* performance goals, detailed study suggests that our performance shortfall is probably related more to the UNIX process architecture and less to our language choice.

After performing code cleanup based on Lennart's suggestions, we prepared for an initial test release of the system during the summer of 2000. That release has been put on hold while we perform integration work with another complex legacy application and improve our monitoring system.

4 Lessons Learned

Working with Erlang over the past year has been educational. We've learned a number of lessons that we think are worth sharing with the rest of the community.

[3] "Imagine your application running for *ten years*. How many uncontrolled processes are you willing to tolerate?" This was a perspective we desperately needed.

4.1 Erlang is Quickly Learned

Even in isolation, a decent programmer can quickly come up to speed on the basics of Erlang, with greater ease than with many other more popular languages. With mentoring, we expect that a new person could be able to understand enough of our existing code to begin making non-trivial contributions in less than a month.

4.2 OTP Isn't So Quickly Learned

Proficiency with the OTP, however, is another matter. In our estimation, there simply isn't sufficient documentation to expect isolated programmers to make decent use of OTP on their own. We often found ourselves making up too many things as we went along. We'd make informal bets that such-and-such a problem had already been solved but we didn't know how. The `erlang-questions` mailing list [erlb] was invaluable — as long as we had a coherent question to ask — but it was hard to know what we might be missing.

If we'd been learning Erlang at Ericsson the same way we learned C, rubbing shoulders with much more experienced programmers and tackling small, self-contained projects, we would have had a much easier experience. Unfortunately, we had to immediately create the architecture for a major application. Without a mentor, the best place for education is by reading existing code of well-written applications. However, without commentary, we would expect the novice Erlang programmer to miss many subtle issues involved in employing it correctly. We certainly did. If mentoring isn't an option, we think a training course in the use of the OTP after the developers have some familiarity with the language is probably wise.

4.3 Erlang Is Good For Both Rapid Prototype and Production Code

Now that we're more proficient with Erlang, prototyping new ideas is a rapid process. Once the prototype is done, it can often be folded into production code with only small modifications.

The language, together with the standard and OTP libraries, provides an extremely useful framework. We're free to consider important operational issues from the beginning, knowing that many mundane details are already taken care of. The process linking concept, together with process supervision trees, is the most valuable, in our experience. The tools for event logging are a close runner-up. And the inter-node message-passing infrastructure is so easy to use it's hard to explain to programmers not familiar with the language.

4.4 Erlang Performs Well

Our first experiences with bulk I/O with Erlang were bad, since the original version of the RPC code moved all data as lists of bytes rather than binaries. Once we modified our code to be binary-friendly, we saw its performance increase by more than an order of magnitude. Without binary data types, though, the language's performance would have been abysmal for our application.

For a single application reading or writing a large file over the network, data throughput measures of the Erlang and C Client Daemons are identical: the performance of both is limited by network latency. When performing multiple concurrent bulk reads or writes through the same Client Daemon, the C prototype is faster, but only by a couple of percentage points. We were pleasantly surprised to find the difference so small. We have found that our performance shortfall, relative to C, is due to a small penalty per byte of bulk data in each RPC transaction, and a large penalty per transaction. We hope that the former can be addressed by a more advanced network driver or other IPC mechanisms, and that the latter can be addressed by further tuning our code.

4.5 We Would Use Erlang Again

Many advanced programming languages are useful for research but have severe deficiencies for production work. Erlang doesn't fit that pattern. Our increased productivity with Erlang more than offset the difficulties of learning the language. Aside from wanting access to UNIX domain sockets and shared memory, the language provided most of the tools we needed to develop the prototype and then expand it into a production-quality system. Indeed, we had difficulties with the packaging and distribution tools largely because they provided more functionality than those in the traditional C/UNIX environment.

We worried that the virtual machine might be too slow for our purposes. Instead, performance has not been a big issue: Erlang's performance is on par with our C prototype. The next big performance increase will come from changing the communication channel between the Client Daemon and its applications, probably using shared memory. If we were to use that scheme in both languages, we expect we would see comparable performance, and we expect the Erlang version would be finished sooner.

If we had to do this all over again, we'd still use Erlang.

5 What Erlang Needs

We've got a long wish list of things we'd like to see in future Erlang/OTP releases: enhancements to the virtual machine, new built-in functions, expanded libraries, more supported platforms, and better documentation. Fortunately, under the Erlang Public License [EPL], the source code is available for us to modify to suit our needs. Many of these things aren't tremendously difficult to do[4], and we may yet implement some of them, but code is almost always nicer if someone else writes and maintains it.

We realize many of these wishes may be fulfilled by the R7 release of Erlang/OTP. However, since we do not have that release at the time this document is being written, these wishes are based on the R6B release.

5.1 Better Documentation of Best Coding Practices

The Erlang book is the best reference we've found for learning the language. However, documentation on the Open Telecom Platform is confined to the reference material in the online documentation [Erla]. In fairness to the current OTP documentation, it is a good reference resource, but it's not a tutorial. You simply need to know where to look and to know if it's the right hammer to pound any particular nail.

Programmers coming from a C/UNIX background are accustomed to an edit-compile-debug work cycle inherited from batch-processing origins. Interactive editors and integrated development environments have accelerated the cycle but have not changed its fundamental character. The interactive interpreter was therefore a little puzzling — how should it be used in daily work? Being unable to learn at the knees of local experienced Erlang programmers, we experimented on our own. Having a detailed "user story" in the documentation, a low-level chronicle of a typical day programming Erlang, would have accelerated this process.

5.2 More, Better, and Faster IPC Mechanisms

Understanding that our desires are biased toward the applications we typically work on, we'd like to see UNIX domain sockets formally supported. We'd also love to dabble with shared memory, though it can be problematic in a system where memory management is hidden from the programmer. We'd love to see a more efficient TCP and UDP driver, one that makes use of the `outputv()` driver interface to allow use of vectored I/O primitives and other efficiency mechanisms.

We understand the portability concerns raised by supporting these admittedly platform-specific features. It can make "write once, run anywhere" code more difficult to write and maintain.[5] Such feature creep presents a slippery slope: what new features are platform-independent enough, or is customer demand great enough? In our opinion, the need for fast IPC warrants their use in the standard Erlang distribution.

As discussed in Section 4.5, Erlang is a surprisingly useful, practical language. Support for additional IPC mechanisms can only encourage other adventurous programmers to develop other Erlang applications that break yet more new ground.

[4] As an example, we've already experimented with having the TCP and UDP drivers allocate their buffers from a shared memory pool.

[5] The Client Daemon runs quite well on Erlang/OTP for Windows NT, despite intentionally ignoring NT during development.

5.3 Better Debugger

The debugger in Erlang/OTP R6 is useful, but it needs enhancements. We would love to see a binding watchpoint feature added. A more streamlined "compile, debug, edit, compile, debug newly-edited code in the same debugger environment" cycle, one that requires fewer mouse clicks, would be nice. Also, it would be handy to save breakpoint settings in a context-sensitive manner, attempting to maintain breakpoint locations despite adding or deleting lines of code prior to the breakpoint.

5.4 Better Profiling Tools

We've spent a good deal of effort trying to understand the performance characteristics of the BEAM VM in general and of our application running within it. The Erlang/OTP R6 profiling tool, `eprof`, is okay at best and utterly inaccurate at worst. We ended up working on a better tool,[6] but the effort has been limited by the VM's process trace output itself: it can fail to mention execution of some short functions, which throws off function call counts and can lead to misattribution of execution time.

The fundamental problem is the lack of a global context for the profiling results. If `eprof` profiling reveals a function to be the tall tent pole within a given process, it may still be insignificant if the profiled process is only a small fraction of the overall runtime. The user-contributed `top` tool [Top] is useful for getting a system-wide view of VM reductions, but it cannot account for reductions made by short-lived processes. Furthermore, there is no accurate correlation between VM reductions and either CPU or wall-clock time.

To give a global context for our performance, we used `gprof` to measure the BEAM VM as a C program. This didn't allow us to directly measure the execution of our program code, but we were able to see the relative weights of bytecode execution, message-passing, garbage collection, and linked-in drivers.

Within the Erlang code, we see the need for both process-oriented and function-oriented profiling. We also need the ability to create a `gprof`-style call graph. Lastly, we need both wall-clock and CPU-clock timing statistics.

5.5 Better String Handling

Being avid Perl hackers, it may be unfair to criticize Erlang for weak string handling features, but we'll do it anyway. The functions found in the standard `string` module are a good base. But Erlang's treatment of strings as lists of bytes is as elegant as it is impractical. The factor-of-eight storage expansion of text, as well as the copying that occurs during message-passing, cripples Erlang for all but the most performance-insensitive text-processing applications.

Erlang's treatment of binaries, by contrast, has so far proven to be a showcase for the language's features without a significant cost in performance. We'd much rather see a `string` library, parser-generator, etc., based on binaries, or on some new binary-like string representation, rather than the current list representation.

5.6 Multiprocessing Support and Memory Usage

Again, on this topic Erlang is caught between a rock and a hard place. On multiprocessing machines, we'd really like to see the virtual machine take advantage of as many CPUs as are available, especially since moving data between Erlang instantiations via IPC currently requires several data copies and is therefore expensive. We see no reason why SMP support would require any change to the language or its libraries. It would require a massive redesign of the virtual machine and possibly large sections of platform-specific code to get the best performance. However, if Erlang wants to be considered for applications like ours on high-end hardware, SMP support is necessity.

We understand that with R7 we'll see the VM able to use up to 4 GBytes of RAM, but, again, for high end applications this isn't enough. We need to be able to run our system on 12+ processor machines with

[6] We were desperate enough to modify `eprof` for greater accuracy and to try to measure both wall-clock and CPU time. See `http://www.erlang.org/ml-archive/erlang-questions/200005/msg00052.html`.

12+ GBytes of RAM. For the foreseeable future, this means running multiple virtual machines per physical server, which is something we'd prefer not to do.

5.7 More Support For Writing Network Servers

Many important Internet applications, such as the Apache web server and the BIND naming daemon, are moving toward a multithreaded programming model. We shudder to think of using C/pthreads to achieve this. With the imminent release of the bit syntax, we see a great opportunity for Erlang to be the premiere language for serious Internet server development.

We'd like to see more support for writing servers providing TCP- and UDP-based protocols. While there are a few sample applications to learn from, more could be done to assist programmers communicating via an IP network to non-Erlang-based clients. Items on this wish list include:

- More documentation and examples of non-trivial clients and servers of popular protocols.
- Allow a listening TCP socket to operate in an "active" mode, i.e. by allowing `gen_tcp:accept()` to send a message to the listening process rather than as a blocking function call.[7] This would allow a `gen_server` to directly accept new connections without blocking.
- Streamline/simplify the process for passing ownership of a newly-accepted socket descriptor process to another process. The most natural way to write a per-session process's `start()` entry point is to have the TCP connection socket as an argument and to give the new process's PID as a return value. However, there is a race between the new process using the socket (e.g. printing a prompt or greeting) and the listener process changing the socket ownership. This race can be resolved with some message-passing for synchronization, but it has occurred often enough in our code to be annoying.
- Avoid needless data copies across the driver boundary.
- The binary syntax will help immensely with network byte-order conversions and (un)packing encoded data structures.

With a few clean-ups and an eye towards attracting a wider audience, we feel that Erlang can make significant strides as a network application language.

6 Conclusion

We were surprised at the extent to which Erlang fulfilled its promises. It took some effort, but it was straightforward for developers to come up to speed on the language, and its use significantly reduced the time it took to produce working code. While we still believe that Erlang has some deficiencies, it has demonstrated itself as a first class prototyping language, and we have no qualms about shipping a production application based upon it. A number of Erlang's features are novel and compelling. As we look at other projects written in C, we often find ourselves thinking about solving their problems with Erlang. While there are a great number of places in which the language and its environment could be significantly improved, Erlang is already a fascinating language that deserves a wider audience than it currently has.

7 Acknowledgments

The authors of this paper would like to thank our management for allowing us to get ourselves into this mess. Thank you as well to those fine folks at Ericsson who came up with Erlang in the first place. Also, a

[7] Some thought should be given to a rate-limiting mechanism for these automatic accepts, although much existing connection accepting code has no such mechanism, either.

special thanks to the folks at Bluetail A.B. for their early encouragement. We especially appreciate the work of Tony Rogvall and his ONC RPC library: it saved us a great deal of work. Finally, we want to especially thank everyone on the `erlang-questions` mailing list for helping a bunch of neophytes come up to speed with their nice language. You've helped us more than you'll probably ever know.

References

[Blu] The Bluetail Mail Robustifier. See `http://www.bluetail.com/products/bmr/`.

[Bro95] Frederick P. Jr. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 20^{th} anniversary edition, 1995.

[Edd] Eddie, an HTTP load balancer. See `http://www.eddieware.org/`.

[EPL] Erlang Public License. See `http://www.erlang.org/EPLICENSE`.

[Erla] Erlang online documentation. Available at the Open Source Erlang distribution site: `http://www.erlang.org/download.html` and in browsable form at: `http://www.erlang.org/doc.html`.

[erlb] The `erlang-questions` mailing list archive. See `http://www.erlang.org/ml-archive/erlang-questions/`.

[KLS86] N. Kronenberg, H. Levy, and W. Strecker. VAXclusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[Lib] Libero, a finite-state machine-based, programming language-independent code generation tool. See `http://www.imatix.com/html/libero/`.

[SMT] SMT, the Simple Multi-Threading kernel. See `http://www.imatix.com/html/smt/`.

[Top] `top-1.0`, a UNIX `top`-like tool. See `http://www.erlang.org/user.html`.