Increasing the reliability of email services

Joe Armstrong
Bluetail AB
+46 8 692 22 00
joe@bluetail.com

ABSTRACT

This paper describes the architecture and implementation of a system for increasing the reliability of existing Internet mail systems. Our system is designed as an add-on to existing software for handling email; rather than designing a new mail system from scratch, we decided to make a system that could be used together with conventional mail programs.

Mail systems are usually built with simple client/server architecture, most often the client software runs on a PC and the server software on a Unix machine. We assume that the client and server communicate using TCP/IP and that mail services are mediated using standard protocols.

In our system we have concentrated on three mail protocols SMTP [10], POP3 [13] and IMAP4 [7]. All client/server communication that we are interested in makes use of these protocols.

To improve the reliability of the client/server system we interpose a "proxy" between the client and server. Seen from the client's point of view the proxy appears as a conventional server and from the server's point of view the proxy appears as a conventional client.

Software in the proxy monitors the client/server communication and can add additional functionality to the system without making any changes to either the client or server software.

Interestingly, the entire system software is programmed in a declarative programming language, Erlang [2]. To the best of our knowledge this is the first time a declarative language has been successfully used for this type of application.

SAC'00 March 19-21 Como, Italy

Keywords

Reliability, email, POP3, SMTP, IMAP4, Erlang, Functional programming.

1. Introduction

This paper describes the architecture and implementation of the Bluetail Mail Robustifier (BMR). BMR is written entirely in the declarative programming language Erlang and runs on conventional Unix operating systems.

We were interested in making an email handling system subject to the following requirements:

1. Down times should be a few minutes per decade.
2. If a mail server fails some other server should take over with a minimum of delay, clients should not notice that the server has failed.
3. It should be possible remotely manage the system, we would like to add or remove mail servers or take them out of service without interrupting the quality of service.
4. It should be possible to upgrade the BMR software itself without stopping the system.
5. In the event of an error in the BMR software it should be possible to revert to a previous version of the software without stopping the system.
6. The system must work together with and improve the performance of existing mail systems.
7. The system should work on existing hardware and run on existing operating systems.
8. The system should be at least as efficient as a conventional imperative language implementation.
9. The system should have a conventional Unix command line interface, and conventional Unix manual pages.
10. The system should have a GUI interface.

Requirements 1, 2 and 3 are typical requirements for soft real time systems (things like Mail systems, ATM systems, Telephone exchanges etc.).

Requirements 6 and 7 were based on the observation that it is very difficult to persuade users to change their hardware or software.

Many operators have large investments in a particular technology and it would be very difficult to persuade them to change.

Our strategy was one of "adding additional components to their existing system which makes their existing system more reliable".

Requirements 9 and 10 were made so as to make it easy for system administrators to learn how to use our system.

Requirements 4, 5 and 8 represented an interesting challenge.

Most existing systems are based on TCP/IP and make use of the SMTP protocol for sending and delivering mail and the POP3 and IMAP4 protocols for reading mail. All these protocols assume a client/server architecture, the client usually runs on a PC in a variety of environments but the server runs in a controlled environment. Large mail systems typically use Sendmail[1] [1] and handle tens to hundreds of thousands of user accounts.

Our system improves the reliability of such systems by interposing a proxy between the clients and the server.

2. ARCHITECTURE

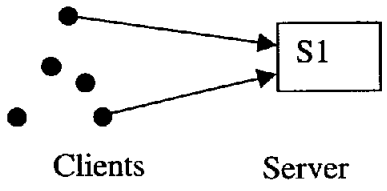

Figure 1: A Single Server

We are interested in conventional clients/server architectures having a single server and multiple clients (figure 1). The server is assumed to be large and complex and the clients small and simple (email client/server architectures fit nicely into this characterization).

Of course, if the server fails, everything fails. To make things more reliable we might add an addition server (figure 2).

The addresses of the two servers must be published in some way[2] so that the clients can find the address of one of the servers.

Unfortunately, this does not solve our problem since if the IP address of one of the servers is cached in one of the clients and if that server subsequently fails, then the client will assume that the server is dead and will not know about the other server which is alive. Moreover, an ongoing session on the failing server cannot be continued on the other server.

[1] We estimate that about 70% of all ISPs use Sendmail running in a Unix environment.

[2] For example, by using multiple DNS records and relying upon DNS round-robin scheduling.

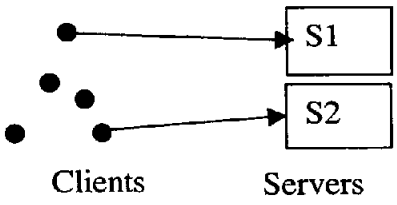

Figure 2: Multiple Servers

Since the clients talk directly to the servers we cannot do any load balancing of the servers, or take the servers out of operation etc. The only mechanism available for building systems with this architecture is to use multiple DNS records and arrange that faulty machines are removed from the appropriate DNS records. Unfortunately it will take some time before these changes have propagated to all hosts that use the server. An alternative is possible to migrating the IP address of the interface from a failed to an active server, but this involves some tricky and non-portable programming techniques and still does not solve the problem of session failover.

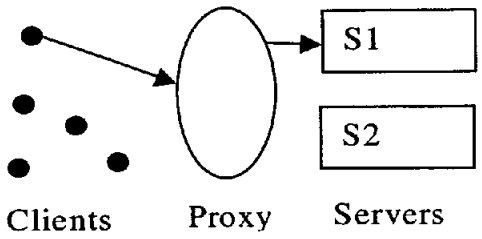

Figure 3:Multiple Servers accessed through a proxy

A better solution is to introduce a *proxy* between the clients and the servers (figure 3). The proxy listens to the same port as the server, analyses the data and relays the data to one of the servers. So, for example, in the case of SMTP our proxy listens to all requests on port 25[3] and when a new mail session starts it chooses a server (we call this a *backend server*) to handle the mail session - thereafter all messages are relayed through the proxy to port 25 of the backend server.

To make the proxy available we simple replace the IP address of the mail server in the appropriate DNS records with the IP address of our proxy.

Again, if the proxy were a single machine, it would become a single point of failure and we would be back at the situation of figure 1 that we were trying to avoid. While we view the proxy as a single *logical* component in our system (as in figure 3) we must always remember that the proxy should be made from at least two physically separated machines - this is show in figure 4 which shows a proxy composed from two machines.

[3] Port 25 is the so-called *well-known port* for SMTP.

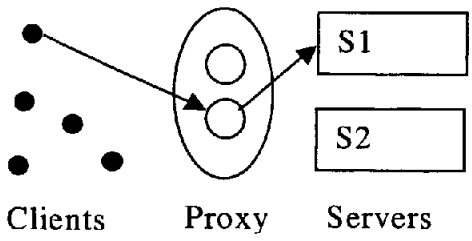

Figure 4: Internal structure of a proxy

From the point of view of a client the proxy serves as an entry point into the mail system - from now on we will refer to this as a *gateway*. While we have only shown two gateway machines and two backend servers, it is easy to imagine gateways with large numbers of machines handling server farms with large numbers of servers.

Give the above architecture we can implement a number of desirable features in the gateway:

- Server maintenance - new servers can be added, removed or upgraded without loss of service to the users. To add a server we simply install and configure the new machine, start the server on the new machine and inform the gateway. To remove a server we inform the gateway that the server is to be taken out of service, the gateway will then ensure that no new transactions are started on this server. Once all transactions on the server have finished (the gateway continuously monitors all such transactions and thus knows when this is true) the server can be removed.[4] Using this facility users can perform planned routine maintenance operations on their machines, or, for example, temporarily remove a machine in order to reboot it etc.
- Load balancing - A small program, which measures the true CPU load, is added to each server. These programs send CPU load information to the gateways, the gateways can then direct new traffic to the least loaded server. Note that existing mail systems often use "DNS round robin" to distribute tasks to a number of different systems this method does not correctly balance CPU loads - our method will correctly balance CPU loads on a number of different machines.
- Overload protection - If the CPU load on one of the servers becomes too high then the gateway can limit the load on this server by directing new requests to an unloaded server. If this strategy fails then we can queue new requests instead of dealing with them immediately, failing this, new requests can be rejected.
- Session failover - In the case were where client/server operations are transaction based, the gateway can record the entire session. If a backend server fails at any point, then a new server can be allocated and the entire session can be replayed against the newly allocated server; this mechanism we call failover and is explained in more detail later in the paper.
- Client monitoring - Sometimes we wish to protect the servers from faulty clients. We could, for example, in the case where a client crashes or there is no client progress for a long time terminate the server session for this client by simulating a client timeout.
- Client screening - we could reject connect calls from particular rogue clients. Certain blacklisted clients might try to damage the servers with various attacks. These can be rejected in the gateway. In fact we implement all the anti-spam proposals outlines in RFC2505 [12].

Operations such as these represent a legitimate "separation of concerns" between what it is reasonable to implement in a server and what it is reasonable to implement in the gateway.

Everything that has to do with the specifics of the server is implemented in the server. Everything that has to do with load-balancing, failure, admission control, maintenance etc. is performed in the gateway.

Following such an approach we arrive at a *generic* gateway that can be used as a front-end for several different application servers.

Having implemented a service in the gateway this service can easily be added to an existing protocol. For example, load-balancing, overload protection etc. applies to all supported protocols. Other features, like failover require specific protocol dependent customizations. These customizations are necessary to implement specific failure mechanisms which provide better behavior than that offered by the default mechanisms.

3. FAILOVER

When a server fails we might want to mask the failure, so that the client does not see the failure. This can be achieved by a "record/playback" technique in the gateway.

At the start of a session the gateway chooses a server. During the session the gateway records all the messages sent between the client and the server. If the server fails then a new server is chosen and all the recorded messages are replayed against the new server. An example of this is shown in figure 5.

In figure 5 a client sends messages **a** and **b** to server one and receives replies to these messages; we assume that the gateway, is transparent, and merely records and relays the messages. After the client sends message **c** we assume that server one crashes and that this is detected by the gateway. The gateway allocates a new server (server 2) and sends messages **a** and **b** to the new server, the replies are discarded. Once the new server has "caught up" message **c** can be sent to the new server. In this way the client will not notice the loss of the server.

[4] This is somewhat simplified.

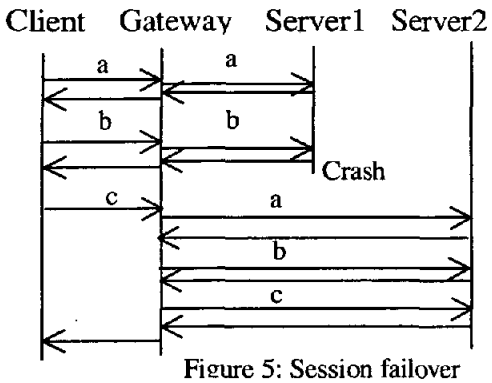

Figure 5: Session failover

The above method provides a *generic* mechanism for session failover but for specific applications we can add protocol specific optimizations to the generic mechanism. An example of such an optimization might be password authentication, if a user is correctly authenticated against server one and the session fails then we could omit the authentication step when replaying the session against server two.

4. HOT STANDBY

The previous section showed session failover using a "record/playback" technique. If we wished to increase the responsiveness of the system we could use a "hot standby technique". In this case copies of the original client to gateway messages are sent to *two* different servers and the earliest response is chosen and sent back to the client. If one of the servers crash then the other server can be used immediately. This is shown in figure 6.

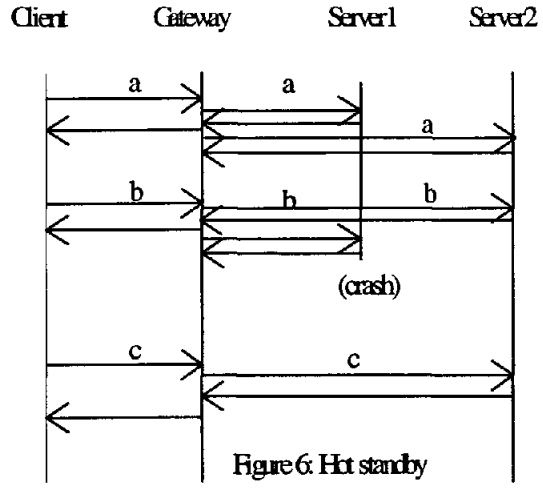

Figure 6: Hot standby

5. SOFTWARE UPGRADE

This section concerns upgrading the software in the BER system itself.

The BER system has several nodes[5]; paradoxically, it turns out to be easier to upgrade software in a distributed system than in a non-distributed system. When upgrading software in a distributed system we have to arrange that all nodes in the system "see" the upgrades at the same time, or, that in the case when they see upgrades to the software at different times that these upgrades are in some sense consistent.

Our upgrade procedures are based on four primitive abstractions:

- *Distribute package* - is an abstract operation that distributes a software upgrade package to all nodes in the system. Distribute always succeeds.
- *Install package* - is an abstract operation that is performed on all nodes in the system and has the result of installing the last package that had been distributed. If the installation procedure on any individual node fails then the software on all nodes automatically reverts back to the previous version of the installed software.
- *Make permanent* - is an abstract operation that "permanentizes" the result of the last *install* operation.

  If any node fails after the install operation has completed but before a make permanent operation has been performed then the software on all nodes is reverted back to the version that ran on the nodes before the install operation was performed.

  After the make permanent operation has been performed then if a node failure occurs then the software is restarted from the latest distributed version.
- *Rollback* - is an abstract operation that can be used after *install* but before *make permanent*. This puts the system back into the state it was before the *install* operation was performed. All nodes revert back to the original program versions.

None of these code handling primitives was available in the basic Erlang system, but can easily be programmed in Erlang. This was implemented in 5 modules and 1,500 lines of Erlang on top of the basic Erlang open source distribution [9].

6. IMPLEMENTATION

The BMR was written in Erlang [2]. Erlang is a declarative programming language that was originally designed for implementing fault-tolerant distributed soft real-time applications. Erlang has been used successfully in a number of industrial products some of these are described in references [4] and [5]. Later examples of these include the Ericsson AXD301 [6] and GRPS systems [8].

To solve problems in this problem domain Erlang combined a number of ideas found in different programming languages and operating systems. Some of the most important ideas embodied in the language were:

[5] at least two

- Write once variables - Erlang has no mutable variables, this eliminates a large class of "pointer errors" found in imperative languages. A less obvious consequence of having no mutable variables is that it is impossible to create circular data structures - which in turn has implications for garbage collection [3],[11].
- Dynamic types - Dynamic types provide type safety, in the sense that a program will never core dump.
- Higher order functions - which greatly improve program clarity.
- Near real time garbage collection - which means that the language can be used for industrial control applications.
- Parallel processes - Erlang provides processes "in the language". Most programming languages have no concept of processes, concurrency being outside the scope of the language and which might be provided by the host operating system. Surprisingly many problems can be modeled in a natural way using sets of parallel processes. State can also be modeled as an argument to a tail recursive perpetual process.
- Hot code swap primitives - Erlang has a number of primitives intended for implementing code-swapping strategies. This is to address the problem of changing code in a running system.

BMR was programmed from scratch in Erlang; it was tested and delivered to our first customer in less than six months. At the start of the period two programmers worked on the project and at the end six.

The system was written in 36,285 lines of Erlang and had 108 modules.

A rough breakdown of the code is shown in table 1, this shows the number of Erlang modules and lines of fully commented Erlang code.

Observe that the generic part of the system was 27,200 lines of code and the application specific code was 9,005 lines. Each new application requiring about 3,000 lines of code.

The cluster software was a mere 4,553 lines of code (18 modules). The GUI was implement as an Erlang server (1 module, 807) lines of code and a Java client (83 classes, 28,400 lines of code).

After delivery the system proved to be extremely stable and was put into live operation by the Swedish Telenordia ISP two weeks after delivery of the first version of the product. At the time of writing the system has never crashed and only a few errors were observed in handling the first million emails.[6]

[6] Caused by non RFC compliant software

Function	# modules	# lines
Generic gateway software	17	4515
Load balancing	6	1950
Cluster	18	4553
Software distribution	5	1500
OAM	14	6545
Data base	4	512
GUI	1	807
Misc library routines	18	2262
Test	9	4556
SMTP	8	4444
POP3	3	1458
IMAP4	5	3183
Total	108	36285

Table 1: Code Statistics

7. CONCLUSIONS

We have demonstrated an architecture that can be used to improve the characteristics of existing installed mail systems. The system can be added to an existing system and improve the overall reliability and manageability of that system.

Our architecture consists of a generic gateway and a number of application specific protocol modules. The average amount of code in each protocol module is about 3000 lines of code.

The control system was programmed entirely in Erlang - the rapid development times and reliability of the product show that declarative languages can used with advantage in areas where they were previous thought unusable.

8. ACKNOWLEDGEMENTS

The work described in this paper was performed by Joe Armstrong, Johan Bevemyr, Martin Björklund, Magnus Fröberg, Joakim Grebenö, Thomas Lindgren, Håkan Millroth, Tony Rogvall, Claes Wikström and Patrik Winroth who are all partners and co-founders of Bluetail.

9. REFERENCES

[1] Eric Allen *Sendmail: An Internetwork Mail Router* in the BSD UNIX Documentation Set, Berkeley, CA: University of California, 1986-1993.

[2] Joe Armstrong, Mike Williams, Robert Virding and Claes Wikström. *Concurrent Programming in Erlang*, Prentice Hall, 1995.

[3] Joe Armstrong and Robert Virding. *One Pass Real-Time Generational Mark-Sweep Garbage Collection,* International Workshop on Memory Management, Kinross, Scotland, September 27-29, 1995.

[4] Joe Armstrong, Erlang - *A survey of the language and its industrial applications.* Proceedings of the symposium on industrial applications of Prolog (INAP96), 16-18 October 1996. Hino, Tokyo Japan.

[5] Joe Armstrong. *The development of Erlang*, In Proceedings of the ACM SIGPLAN International Conference on Functional Programming, pages 196 - 203, ACM Press, 1997.

[6] AXD 301- *A new generation ATM switching system* Ericsson Review number 1, 1998.

[7] M. Crispin Internet Message Access Protocol - Version 4 RFC1740. December 1994.

[8] GPRS99 *GPRS-General packet radio services* Ericsson Review number 2, 1999.

[9] *Open Source Erlang* - available from **http://www.erlang.org/**

[10] J. B. Postel *Simple Mail Transfer Protocol* RFC821. August 1982

[11] Dan Sahlin and Kent Boort. *A Compacting Garbage for Unidirectional Heaps* 9th International Workshop on Implementation of Functional Languages. St. Andrews, Scotland, September 10-12, 1997.

[12] G. Lindberg. Anti-Spam Recommendations for SMTP MTAs RFC2505. February 1999

[13] J. Myers and M. Rose. *Post Office Protocol - Version 3*, RFC1725. November 1994.