Generic Load Regulation Framework for Erlang

August 3, 2010

Ulf Wiger
Erlang Solutions Ltd
ulf.wiger@erlang-solutions.com

Abstract

Although Telecoms, the domain for which Erlang was conceived, has strong and ubiquitous requirements on overload protection, the Erlang/OTP platform offers no unified approach to addressing the problem. The Erlang community mailing list frequently sports discussions on how to avoid overload situations in individual components and processes, indicating that such an approach would be welcome. As Telecoms migrated from carefully regulated single-service networks towards multimedia services on top of best-effort multi-service packet data backbones, much was learned about providing end-to-end quality of service with a network of loosely coupled components, with only basic means of prioritization and flow control. This paper explores the similarity of such networks with typical Erlang-based message-passing architectures, and argues that a robust way of managing high-load conditions is to regulate at the input edges of the system, and sampling known internal choke points in order to dynamically maintain optimum throughput. A selection of typical overload conditions are discussed, and a new load regulation framework – JOBS – is presented, together with examples of how such overload conditions can be mitigated.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Reliability, availability and serviceability

General Terms Regulation, Performance

Keywords Erlang, Performance, Throughput, Regulation

1. Introduction

The Erlang programming language (Armstrong 2007) is predominately used in server-side applications and various forms of messaging gateways. These systems are often exposed to bursty traffic, and need a strategy for coping with overload conditions, not least in order to provide critical service in the event of disasters and other unusual events (See Figure 1). As Erlang is a highly concurrent, message-passing language, overload conditions have much in common with congestion problems in communication networks and other traffic engineering systems. Indeed, Erlang-based applications are often also part of such communication networks, and must take responsibility for delivering predictable throughput, in order not to become a congestion point and cause even greater problems.

Erlang'10, September 30, 2010, Baltimore, Maryland, USA.

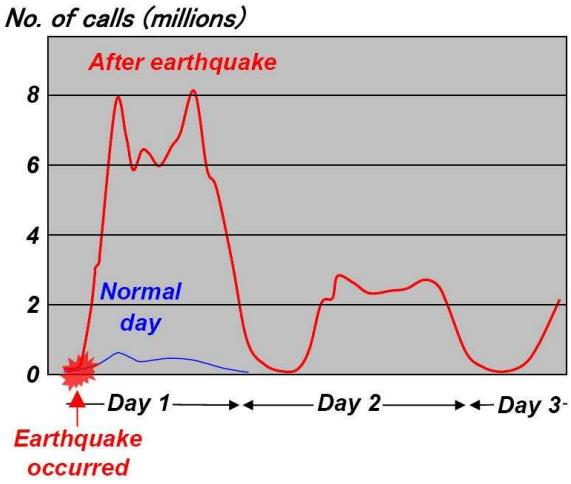

Figure 1. Land-line telephone traffic congestion after Hanshin-Awaji Earthquake, see (Kitaguchi and Hamada 2008).

Figure 2. Result of high ranking on Reddit.com, see (Wilson 2008).

While the telecoms domain has changed since Erlang was invented – nowadays, telecommunication is mainly regarded as a specific form of Internet multimedia – Erlang-based applications are still often used in systems serving human communication patterns – instant messaging, Voice over IP, search and document rating networks. As Web users on a global scale can "flock" towards information, online services can be exposed to extremely bursty input loads (See Fig 2).

Some notable Erlang-based products have boasted impressive resistance to overload (see Fig 3 and Fig 4), but not much has been written about how it is done.

In the following sections, we will list some typical problems that can arise in Erlang-based applications due to overload, and describe some common mitigation strategies. We will then outline a general approach to load regulation of Erlang programs, and describe a

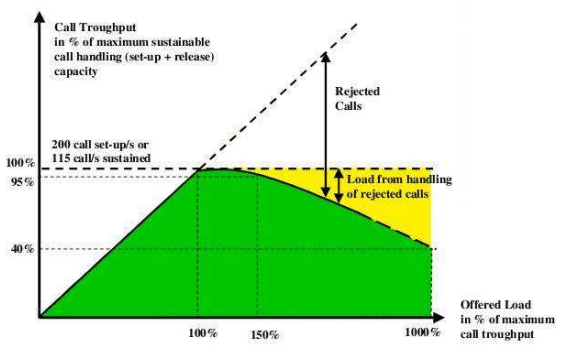

Figure 3. AXD 301, throughput under load, see (Wiger 2001).

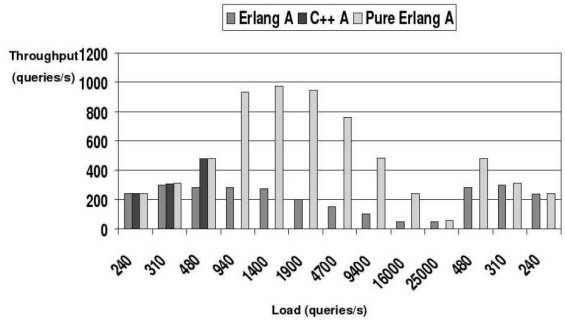

Figure 4. Dispatch Manager, increasing and then decreasing load, C++ vs Erlang; in this test, the C++ application dies after 480 queries/s, see (Nyström et al. 2008).

framework designed to offer generic support for load regulation, addressing these problems.

A perhaps unusual approach in this paper is to compare regulation of Erlang-based systems with the problem of achieving Quality of Service (QOS) for multimedia traffic over IP networks. An intuition for this might be that Erlang, being a concurrency-oriented language, supports the building of message-passing systems, somewhat similar to that of a communication network. Erlang was designed for soft real-time, where response times are usually quite good, but no hard guarantees are given. This could also be said for IP networks with DiffServ QOS (see (Peuhkuri 2010)).

Our hypothesis is that lessons can be learned by studying the migration of voice traffic from dedicated circuit-oriented networks to packet-oriented best-effort IP networks, with an intemediate step based on ATM – a packet-oriented technology capable of supporting guaranteed bit rates. ATM lost to IP much due to its higher complexity, cost of interfaces and failure to provide high-speed interfaces for the core network in a timely fashion (see (Gray 2000)).

Our proposed approach is to consign load regulation to the edges of the system, and to be very careful with adding flow-control measures in core components. The reasoning behind this is further expanded in subsection 3.3, Stateless Core.

2. Common Problems

2.1 Active sockets

All communication between Erlang processes and external entities is done through `ports`. Ports behave more or less like normal

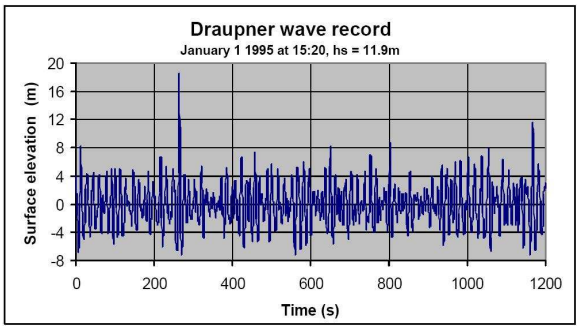

Figure 5. Time history including a possible freak wave event, Draupner platform, North Sea Jan 1 1995, see (Haver 2003).

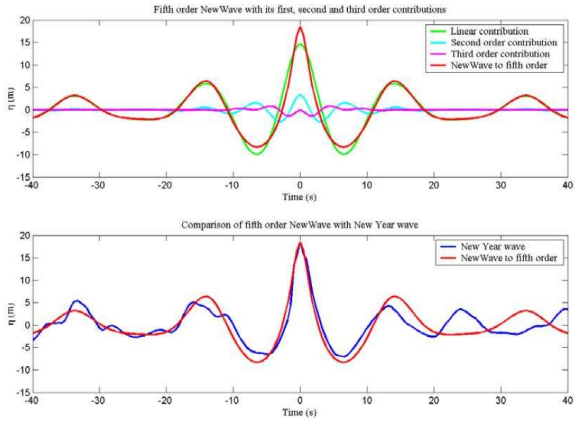

Figure 6. Approximating the Draupner monster wave using a 5^{th} order non-linear equation, see (Taylor et al. 2006).

processes, in that they send a message to the "port owner" process when there is incoming data, and they are instructed to send data by sending them a message in return.

Ports are by default interrupt-driven, but in the case of `inet` ports (sockets), they can operate in different modes:

- `{active,true}` – incoming data on the socket is immediately passed to the port owner.
- `{active,false}` – nothing is sent until the port is instructed to do so.
- `{active,once}` – the port sends one message as soon as there is available data, then reverts to `{active,false}` mode.

In the case of TCP sockets, keeping the port in `{active,true}` mode means that senders will immediately be free to send more data. This can be fatal if the Erlang system is not able to process incoming data fast enough. It is generally considered prudent to keep the sockets in `{active,false}` or `{active,once}` mode. It should also be noted that the low-level POSIX socket API is "passive", in that data must be explicitly read from the buffer.

2.2 Memory Spikes

Garbage-collected languages are known to sometimes exhibit bursty memory allocation behaviour, and Erlang is no exception. Certain job types may be more demanding for the garbage collector, e.g. by building large terms on the process heap, causing repeated garbage collection sweeps that fail to free up data.

Traditionally, this has been a well-known, but reasonably manageable problem. Erlang offers the ability to trace on garbage collection events in real-time, making it possible to quickly identify processes that cause memory allocation bursts.

However, multi-core architectures introduce the possibility of multiple schedulers injecting this sort of behaviour simultaneously, potentially causing lethal memory allocation peaks. We have seen this happen during load testing of otherwise robust message-passing systems. The phenomenon is non-deterministic and can require hours of load test to discover. For lack of a better analysis, we called them "monster waves" (see Figure 5) although we do not yet know exactly what causes it. If the analogy would turn out to be appropriate upon further study, it would be highly interesting to explore ways to predict such patterns before they occur (see e.g. Figure 6).

2.3 Asynchronous Producers

Message passing in Erlang is asynchronous. If a synchronous dialogue is needed, this must be accomplished with a request/reply pair of messages. Reply messages are necessary if the client needs confirmation that the request has been handled, but it also has a flow-control effect.

Some behaviours in Erlang are primarily asynchronous. The OTP behaviour `gen_event` is one example. An event notification is done with an asynchronous message, whereas the processing of the event involves a sequence of (synchronous) calls to user-defined handler functions. The `error_logger` process typically pretty-prints each event and logs it to disk, spending far more effort on each event than the sender does. Therefore, it is quite easy to overwhelm the error logger and potentially killing the whole system. It should be noted that this can happen without any particular wrongdoing on the part of the user, even though it has happened that unsuspecting developers have triggered this problem by relying on the `error_logger` for debugging output

2.4 Unnecessary Serialization

Getting the right granularity of concurrency as well as the right balance between synchronous and asynchronous communication is quite difficult, especially for the novice programmer. One example of a beginner's error is to think that each component must have its own process – typically a `gen_server`. In large organizations, this model can be favoured for organizational reasons – it is easier to perform unit tests if you have your own process. As noted above, it also introduces natural flow control, as the caller necessarily must wait until the work is done.

This style of programming can easily lead to excessive serialization, and the introduction of bottlenecks. We are reminded of Amdahl's argument (Amdahl 1967), stating that "the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." Simply put, while excessive serialization may give benefits in terms of flow control, it is likely to come at the expense of throughput.

2.5 Excessive Contention

There are very few shared data structures in Erlang. `ETS` tables are a form of off-heap storage, either hash tables or b-trees, which can optionally be shared among all processes in the system. This means that the `ETS` tables must be protected internally by mutexes.

`ETS` tables became ubiquitous early on, when contention was not a problem (there was only one cpu, and one scheduling thread), and putting the data in `ETS` was often a way to speed up processing compared to using functional data structures on the process heap. But with the introduction of multi-core and multiple schedulers, lock contention has significantly altered this relationship.

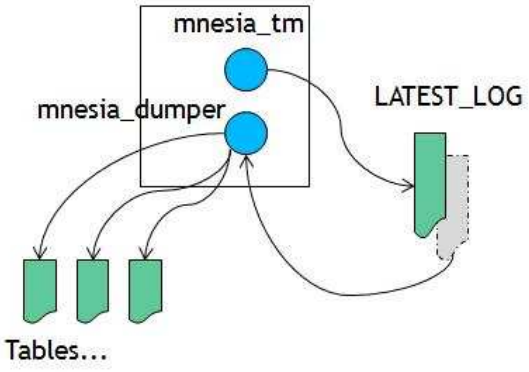

Figure 7. Dumping of Mnesia's transaction log.

Another area where contention can become a problem is `mnesia` transactions. `Mnesia` employs deadlock prevention, which is a reasonably scalable method of avoiding deadlocks in a distributed environment. However, deadlock prevention introduces the risk of false positives (it allows lock dependencies to flow only one way, thus making deadlocks impossible, but also punishing some transactions that were never in danger of deadlocking). The greater the number of concurrent transactions operating on the same data set, the greater the risk of such false positives.

2.6 Large Message Queues

Erlang's support for selective message reception is a great strength, but the implementation – pattern-matching over a single message queue - has rather poor complexity. Behaviours such as `gen_server` normally pick the first message in the queue (a constant-time operation), but in the handling of a message, it is quite possible that the server communicates with other processes and resorts to selective message reception. If the server cannot keep up with incoming requests, each time it performs a selective receive, it must scan all messages in the queue – an operation that becomes more costly the more it falls behind.

In OTP R14B, an optimization enables functions like `gen_server:call()` to run in constant time, regardless of the length of the message queue, but processes that use selective message reception in places where they also receive 'normal' messages, will not be able to benefit from the optimization (see e.g. subsection 2.7).

2.7 Mnesia Overload

Mnesia supports checkpointing to disk of data that is supposed to be persistent. Specifically for data that resides both in RAM and on disk, at transaction commit, Mnesia logs the persistent operations to disk by appending them to a commit log. At periodic intervals, Mnesia reads the commit log (starting a new log for future commits) and distributes the changes into table-specific logs, periodically merging those logs into the actual table image (see Figure 7).

This procedure is quite fast, as it relies entirely on streaming data to and from the disk, but on occasion, the next log dump may be triggered before the previous log dump has finished. When this happens, Mnesia will report that it is overloaded.

Another form of overload in Mnesia is when a transaction manager is not able to keep up with updates originating on remote nodes. This can cause the message queue to grow in the transaction manager process, slowing it down and causing it to fall even further behind (see subsection 2.6). Mnesia detects this and reports that it is overloaded, but the 'overloaded' event is only issued on

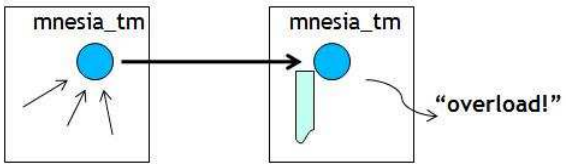

Figure 8. Overload of the mnesia transaction manager.

the node where the overload was detected – not on the nodes where the load originated (see Figure 8).

3. Regulation Strategies

Before looking at concrete techniques for mitigating the problems mentioned, we should pause to consider what kind of system an Erlang-based program comprises from a regulation standpoint – if indeed such a distinction can be made. There are several possible strategies for regulating the work of a system.

3.1 Feedback Control

In the textbook "Applied Optimal Control and Estimation" (see (Lewis 1992)), Frank L. Lewis gives the following description: "Feedback control is the basic mechanism by which systems, whether mechanical, electrical, or biological, maintain their equilibrium or homeostasis. [It] may be defined as the use of difference signals, determined by comparing the actual values of system variables to their desired values, as a means of controlling a system. An everyday example of a feedback control system is an automobile speed control, which uses the difference between the actual and the desired speed to vary the fuel flow rate. Since the system output is used to regulate its input, such a device is said to be a *closed-loop* control system."

Feedback Control saw its first applications in ancient Greece, where in 270 B.C. Ktesibios invented a float regulator for a water clock. The regulator kept the water level in a tank at a constant depth, which resulted in a constant flow of water through a tube at the bottom of the tank, filling a second tank at a constant rate. The water level in the second tank could then be used to measure elapsed time.

The development of Feedback Control stalled when Bagdad fell to the Mongols in 1258, but was revitalized during the Industrial Revolution, and has now evolved into a discipline based on mathematics (modern control theory) and engineering. Given that a system can be described in terms of a mathematical model of the input signal and the desired output, a feedback circuit can be constructed. Regulation can then be tuned by combining suitable measures of proportional, derivative (change-oriented) and integral (stabilizing) feedback (see also (Theorem.net 2001)). Special care must be taken to ensure that the feedback loop doesn't in fact make things worse, e.g. by introducing oscillations in the system. A large part of literature on how to do this deals exclusively with linear and deterministic systems.

To establish a secure footing in control theory, we should require deterministic mathematical models of the processes we wish to regulate, but this seems problematic given our context. Not only do such models generally not exist for the problems we wish to attack, but requiring such models would introduce a very high threshold for most Erlang programmers. This view also finds support in (Welsh et al. 2001).

However, by constructing a regulation framework which provides the means to measure and control throughput rate and load characteristics, we might well be able to lay a foundation on which we can apply simple "engineering-style"[1] feedback control, and perhaps add more sophisticated regulation techniques later on.

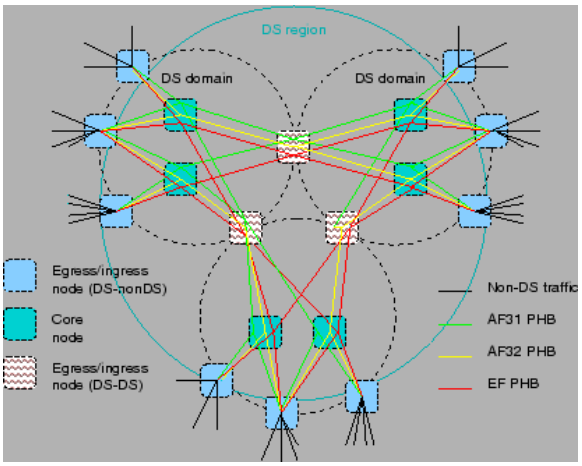

Figure 9. Principle of a DiffServ network, see (Peuhkuri 2010).

3.2 Frequency Regulation

Depending on the characteristics of the system, it may be desirable to regulate based on frequency. One might want to control that the system outputs work at a given rate, or estimate the frequency of incoming requests.

Output rate regulation is relatively straightforward: the very simplest form would be to start an interval timer, which periodically sends a message to a producer or performs a given task.

Frequency estimation is a bit more difficult. It is common to assume that the incoming traffic follows a random distribution, and can be modeled as a Poisson process (see e.g. (Welsh et al. 2001) and (Wikipedia 2010)). In the case of estimating arrival frequency for the purposes of load regulation, we ought in particular to look at non-homogeneous Poisson processes – where the rate is not expected to be constant over time. There is an OTP library, `overload`, which implicitly estimates request frequency (see 4.2).

3.3 Stateless Core

Along with increasing demand for performance-critical services on top of IP networks, much effort has gone into engineering Quality Of Service guarantees on top of TCP/IP. These efforts have suffered from much the same problems that plagued ATM, which led to the currently dominant trend of using IP routers in the core network with only minimal support for packet classification and prioritization. This model is known as the DiffServ model (see (Peuhkuri 2010) and figure 9), or the stateless-core model, in contrast to the Integrated Services (IntServ) model, which dictates per-flow handling throughout the network. The word 'stateless' in this context refers to the router's knowledge – or lack thereof – of individual packet flows.

One of the arguments favouring a stateless core network is the amplification problem, where errors in the core network have much greater impact than errors in the edge network. This has particularly influenced network administrators to favour simple solutions (see (Bell 2003)).

We believe that this dynamic is vital even in software component design. Adding logic to handle congestion issues in core software components leads to increased complexity. Also, as these

[1] as in: relying more on intuition and pragmatic experimentation than a mathematical model

components tend to have little information about the particulars of each application (due to their generic nature), it may be necessary to support different regulation scenarios (see e.g. 6.1).

To the greatest extent possible, we advocate that generic components should stay neutral to load regulation strategies, and focus on being as simple and generic as possible. We do propose adding diagnostic functions allowing users to sample performance characteristics. This is important not least for debugging, but can also be used e.g. in connection with the JOBS framework.

4. Specific Techniques

4.1 Worker Pool

Worker pools are commonplace not least in programming languages that depend on POSIX threads for concurrency. As creating such a thread is quite a heavy operation, it is usually better to create a thread pool at startup, and then pick the first available thread for a unit of work.

In Erlang, this technique is reasonably common in connection with socket servers (e.g. HTTP servers). A predefined number of acceptors can be created, and indeed, all can call accept() on the same socket simultaneously. An incoming request is routed to one of them, and this worker can either acknowledge immediately to an acceptor pool manager, allowing a new worker to be created, or it can do so once it has finished serving the request. The latter would serve as a form of overload control, whereas the former would primarily increase throughput.

4.2 Request Frequency Estimation

The Erlang/OTP framework has a library module called `overload`, which was developed for the very first commercial Erlang-based product, Mobility Server. It is a server which simply keeps track of the frequency of requests to perform work. It grants requests, as long as the frequency stays below a predefined threshold, and then denies requests above that limit (using a hysteresis function to keep denying requests until the frequency has come well below the threshold again). The server uses a simple exponential formula to calculate the frequency (see (Ericsson 2010b)):

$$i(n) = K + i(n-1)e^{-K(T(n)-T(n-1))}$$

where *i* is the intensity (frequency), K, or 'kappa', is a constant which regulates how quickly the calculation responds to changes in intensity[2], and $T(n)$ is a timestamp signifying the time of event n.

While simple, the module lacks the ability to discriminate between different types of work. Also, in many cases, it is not sufficient to simply 'deny' a request, so some form of queueing system is typically needed. If we have a queue, the rate of jobs entering the queue is not nearly as interesting as the number of jobs leaving it, and we don't need an exponential distribution to regulate that. Setting a timer corresponding to the period $\frac{1}{f}$, and dispatching the next job(s) when it fires is sufficient[3].

4.3 Credit System

Credit systems are often combined with queueing semantics. The basic principle is that each client is assigned a quota of sorts, and is allowed to do work until the quota is exhausted; at this point, it must wait until it is assigned a new quota. Erlang's reduction-counting scheduler works according to this principle, and the TCP request window is also a form of credit system.

At the Erlang application level, one may implement a form of quota system with a `gen_server` and simple counters, e.g. using `ets:update_counter/3`. A process wanting to start a task, makes a synchronous call to the quota server, which grants permission if there are credits left; otherwise, it saves the request and responds to it only when already running tasks have finished and returned credits. One complication is that one must handle the case where tasks do not finish in an orderly fashion. If each task runs in a dedicated process, credits can be returned when the process terminates.

A clear advantage of the Erlang programming model with lightweight, pre-emptively scheduled processes, is that the process asking for permission can simply wait in a (blocking) synchronous call, assuming that the process is dedicated to one task, or sequence of tasks.

4.4 Smoothing

Subsection 2.2 dealt with memory spikes, especially in multi-core systems.

A queue-based regulation system may have the effect of smoothing out bursts of jobs, reducing the risk of "monster waves". In our own experience, identifying jobs that resulted in large message heaps (using `erlang:system_monitor/2`) and then regulating them with a combination of a queue and a counter-based credit system (see subsections 4.6 and 4.3) can mitigate this problem.

4.5 Back-pressure

Subsection 2.1 dealt with the possibility to keep network sockets in passive mode. From a load regulation standpoint, it is essential to make use of this technique in order to benefit from the transmit window in TCP.

If the clients are cooperative, or able to use some custom protocol, back-pressure can be achieved through a credit system involving the client (see subsection 4.3). However, in many cases, using standard protocols is part of the product requirements, and few of these protocols support any form of credit or backpressure mechanism. See (Wang 2010) for an interesting discussion on the subject.

Even internally to the system, it may be critical to monitor the status of potential bottlenecks and feed back information to the load regulator, allowing it to adjust the throughput rate.

4.6 Job queue

Erlang-based systems are replete with queues: scheduling queues, message queues, I/O buffers, etc. As a rule, the Erlang runtime system handles all queueing of processes and messages, with the exception of process message queues, which are a fundamental concept in the language.

We can see that queues are a common way to achieve fairness and balance among multiple tasks, and it makes sense to use them at the application level as well as for low-level units of work. We can thus model a type of queue that deals with more coarse-grained tasks. We call this type of queue a "job queue". At the moment, we do not care about how this queue is implemented, only how it is used.

In short, we classify incoming requests as the beginning of a specific type of work (a "job"), and enter it into the appropriate job queue. At some point, we extract the job from the queue and either accept it or reject it. Depending on job type, the queue may be configured with a timeout, at which point we must reject the job, as it may not make sense to continue. An example of this is where protocol timers define a maximum wait time.

For time-limited requests, it may make sense to handle jobs in LIFO order rather than FIFO. During normal operation, it makes no difference in practice, as the queues are expected to be short, and all job requests handled within specified time limits. However, during bursts of incoming traffic, the last entries may be those most likely to complete, as they have used up less of the alloted time waiting in line. Given that the position in line is stochastic anyway, there is

[2] The time it takes for a change in intensity is approximately $\frac{1}{K}$.

[3] We would have to adjust for timer drift due to scheduling delays and the time it takes to perform the actual dispatch.

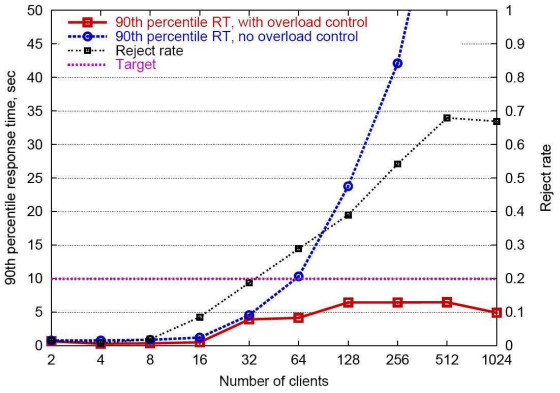

Figure 10. SEDA: Response times with and without overload control, see (Welsh et al. 2001).

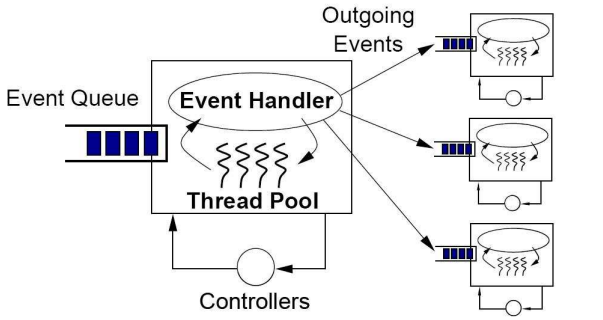

Figure 11. A SEDA Stage, see (Welsh et al. 2001).

not much use in being "fair" to those jobs that happened to arrive at the head of the burst; they are no more "deserving" of being served than the last entries.

Why bother at all about the order, then? Depending on queue implementation, there may be a significant cost difference between the two alternatives. If the queue is implemented as a list, LIFO-style enqueue/dequeue are both O(1), whereas with e.g. the (FIFO) `queue` module in OTP, enqueue is still O(1), but dequeue is amortized O(N). Presumably, lower queueing overhead contributes to increased capacity in the number of jobs that can be handled, which would seem to be the most fair overall.

A LIFO list-based queue also has the advantage that if there is a point where remaining jobs can be discarded (e.g. because the client has already given up waiting), this can be done by simply discarding the tail of the list. On the other hand, the queue will be subject to garbage collection sweeps, which incur a cost proportional to the amount of live data for each sweep.

A queue can also be implemented as an `ordered_set` ETS table, sorted on timestamp, in which case FIFO and LIFO have the same complexity – O(logN). This sort of queue has worse constant factors, but predictable behaviour overall.

In both cases, one should queue only a reference to the requested work, not the full set of input parameters, in order to avoid unnecessary copying.

4.7 The SEDA framework

The Staged Event-Driven Architecture (SEDA), described in e.g. (Welsh et al. 2001), (Welsh and Culler 2001) and (Welsh et al. 2000) is a comprehensive, and highly interesting example of the use of job queues to achieve high throughput and load tolerance in Internet services (see Figure 10).

We believe that the findings from the SEDA framework corroborate our own, and it is perhaps worth noting that we have reached very similar conclusions, while we have approached the problem from different angles.

An important difference between SEDA and the JOBS framework is that SEDA was designed with 'conventional' languages (such as Java and C++) and concurrency models in mind. The stated premise was that there are two generally accepted models for implementing concurrency in high-availability Internet service frameworks: POSIX threads and event-driven programming (or a combination of the two).

"The key idea behind our framework is to use event-driven programming for high throughput, but leverage threads (in limited quantities) for parallelism and ease of programming. In addition, our framework addresses the other requirements for these applications: high availability and maintenance of high throughput under load. The former is achieved by introducing fault boundaries between application components; the latter by conditioning the load placed on system resources." (see (Welsh et al. 2000), pg 2).

While this is a perfectly acceptable premise. Erlang offers significantly different characteristics. The main criticism that POSIX threads are resource hungry and have scalability problems, and that event-based programming can get complex, are not as relevant in an Erlang context, as Erlang's processes handling is quite lightweight and scalable.

In a sense, a significant part of SEDA attempts to provide benefits that are already available to the Erlang programmer. Our assumption is that the load regulation framework should not try to enforce a concurrency model on the programmer, as it would not likely be an improvement over the existing concurrency model in Erlang. For this reason, the JOBS framework is quite similar to a subset of SEDA – the actual request queueing and dispatching component, called a SEDA Stage (see Figure 11). Of course, with a thread pool of size 1, the SEDA Stage also looks quite similar to an Erlang process.

We suggest that the SEDA framework can be a great source of inspiration on how to evolve the JOBS framework, but we imagine a much more coarse-grained division into load-regulated blocks – primarily doing admission control at the input edges of the system, and otherwise relying on Erlang's concurrency model and design patterns to enable high throughput.

5. Load Sampling

The following subsections will give examples of common sampling points, for determining overload conditions. Which methods to choose will vary depending on the characteristics of the system as well as the Erlang/OTP version used.

5.1 Run queue

In a single-core Erlang VM, the run queue is a fairly reliable indicator of high load. The run queue is normally zero, or close to zero in a system that is not overloaded. Sampling the run queue is easy, and a very cheap operation (`erlang:statistics(run_queue)`).

On a multi-core system, this indicator is likely less practical, as sampling the total run queue length involves acquiring a global lock on all scheduler run queues. It is possible that future implementations will loosen the atomicity requirement, making the function more scalable.

5.2 Memory

There are a number of memory-related indicators available in the Erlang runtime system. It is important to understand that the sam-

pled values are rough estimates, and do not reflect things like memory fragmentation, etc.

Depending on the nature of the application, it might be best to use either total memory, the size of the shared binary heap, or the total size of the process heaps as a main indicator. Profiling of the system is needed to determine the most suitable choice.

5.3 ETS Tables

The total amount of data in ETS tables can sometimes be a reliable indicator of the "load" of a system. The total amount of data can either be sampled using `erlang:memory(ets)` or by summing the results of `ets:info(Table, memory)`. The former is more accurate, according to the OTP team.

It might sometimes be the case that a single table, or a subset of all tables, best reflect the system load. Again, profiling is needed to determine this.

5.4 Number of Processes

In principle, Erlang can handle a huge number of concurrent processes – up to 268 million (see (Ericsson 2010a)). However, due to memory constraints (pre-sizing the process table), the configurable hard limit is usually set much lower than that, and the default process count limit is currently 32768. This means that it is quite possible for a system to run out of processes. In some systems, the number of concurrently executing processes might also be a good measure of the load.

In the old days, one could get the current number of processes by calling `length(processes())`, which has O(N) complexity. The function `erlang:system_info(process_count)` yields the same result, but in a much more efficient way. The function `erlang:system_info(process_limit)` might also be useful.

5.5 Message Queue Size

Subsection 2.6 dealt with the problems of large message queues. It is possible to sample the message queue length through `process_info(Pid, message_queue_len)`. Rather than sampling all message queues in the system, one might want to single out a few strategic processes. In most cases, the BIF `erlang:system_monitor(Pid, [{large_heap, Sz}])` should be able to detect large message queues indirectly, as the message queue is part of the process heap. However, if a significant portion of the data in the queue are binaries (which are stored on a separate binary heap), such queues may go unnoticed. The problem with large message queues depends on the number of messages, not the amount of data.

5.6 Response Times

Increasing response times is often a fairly good indicator of overload problems. One of the untold stories from the AXD 301 days is that it was made to sample the response times of neighbouring nodes, since some of them (of a competing brand) had a tendency to die when exposed to more than 150% load, e.g. while re-establishing connections after a link failure. The AXD 301 was capable of slowing down, queueing up traffic, so that other nodes in the network wouldn't topple over from overload.

In this case, sampling response times had a smoothing effect (see subsection 4.4), but it would also be possible to detect local overload by measuring one's own response times. This could be done by measuring the difference between input and output rates, or by simply timing some strategic `gen_server:call()` operations in the system.

6. Rules of Thumb

6.1 Regulate at the Input Edges

In section 3.3, we described the current trend in IP networking to rely on minimal classification in the core network, and relegate regulation to the edges of the network.

Our experience from regulating Erlang applications is similar. It is costly and awkward to implement overload controls in each core component. It is generally better to make them as fast as possible, and insert minimal logic to allow them to indicate if they are getting overloaded. One could imagine complex components like mnesia using a load regulation framework for tasks like transaction log dumps.

In fact, this could also serve to illustrate our point: there is an option, `dump_log_load_regulation true | false`, which makes it possible to have mnesia run the transaction log dumps at lower priority. This can make sense in e.g. a telecoms system, if mnesia is used primarily for configuration data, and performance-critical processes do not write to persistent tables at all. In a system where writes of persistent data are frequent, it makes no sense to perform this type of regulation, as the log dumps must rather be as fast as possible in order to keep up with the traffic load.

6.2 Regulate Only Once

An additional problem with regulating in generic components is that it becomes difficult to know if a job has already been regulated. If classification is done at the input edge, where a unit of work enters the system, it is possible to classify and regulate only once, thereby reducing overhead, and allowing for more predictable behaviour.

However, if a job crosses node boundaries inside the system, it is wise to regulate also on the receiving node. One should bear in mind that the job has now been accepted by the system, and work has already been invested in it. Therefore, it should receive higher priority (e.g. higher throughput) than new job requests entering the system on that node.

With a queue-based regulation framework, classifying the request as an already accepted request would involve putting it in a separate queue.

There may be other cases where it makes sense to regulate multiple times, and we can consider that the SEDA approach is to view multiple regulation steps as building blocks to be combined (see (Welsh et al. 2001)).

6.3 Regulate All Types of Work

It is easy to make the mistake of regulating only the most important jobs. However, ensuring that only the important jobs submit to load regulation and queueing, while other jobs are allowed to run unrestricted, essentially introduces priority inversion.

Therefore, all significant jobs should submit to load regulation. By significant, we mean that the cost of performing load regulation on the job is insignificant compared to the cost of performing the actual work.

6.4 Rejectable vs Non-Rejectable Jobs

A very common type of activity for Erlang-based applications is session establishment (phone calls, logging in to chat groups, etc.) We may note immediately that a request to establish a session may be rejected, whereas it makes little sense to reject a request to end a session. Even in the case where the dialogue times out and the user gives up waiting, we should remember the request to terminate the session, thereby freeing up resources in the system.

In other cases, regulatory requirements may forbid us to reject even session establishment requests. For example, it is not accept-

able for a telephone switch to reject emergency calls. They should be serviced even under extreme overload.

In other words, we must be able to distinguish between rejectable and non-rejectable jobs and treat them accordingly.

6.5 Reject Window

There are three cases to consider once we have decided not to service a request:

1. The client does not expect a reply
2. We have time to send a reject message to the client
3. The client has already given up waiting for a reply

It is considered good form not to send reject messages after the expiration of the protocol window during which the client expects to receive such a message. If we hold requests in a queue until they can be served, we should be able to detect when jobs are too old to even be rejected, and simply discard them all.

6.6 Priorities/Weights

Erlang has some support for process priorities, but it is generally believed in the Erlang community that priorities should be used sparingly or not at all.

In our approach, we choose to prioritize work units rather than processes. It is possible in this model to assign different weights to different jobs, and this may be advantageous for a few reasons:

- Some jobs may be defined as more important than others (e.g. emergency calls vs normal calls)
- We may already have invested processing time in a job. This could happen if part of the job is handled on one node and the rest on another. The job should be given greater priority on the other node, compared to jobs we have not yet committed to.

6.7 Do not optimize for fair weather

It is quite easy to conclude that load regulation must be as cheap as possible, and start designing schemes that perform very well under normal conditions. An example of such a scheme is a credit system, where a pool of credits is periodically refilled, and jobs are allowed to execute immediately as long as there are credits in the pool.

Such a scheme may have excellent properties for other reasons, but it is important to keep in mind that it usually serves no purpose to save CPU cycles when there is plenty of CPU capacity to spare. As justification for this, consider that the nature of Internet service systems, or telecoms systems, is to serve the offered traffic, and nothing else. Any surplus capacity remains unused, and should merely be viewed as spare capacity in preparation for traffic peaks. An application running on a multi-purpose server, on the other hand, cannot easily make this assumption.

Load regulation needs to be the most efficient when there is overload, and resources are scarce. Queueing models tend to display this behaviour; they seem unnecessarily expensive when there is no overload (and therefore no need to regulate), but become increasingly effective when throughput is high, because of caching and batching effects.

7. The JOBS Framework

7.1 Introduction

The `JOBS` framework brings together experiences from several different load regulation techniques, and tries to facilitate most of the techniques above.

7.2 Architecture

Figure 12 illustrates the process model of the JOBS framework.

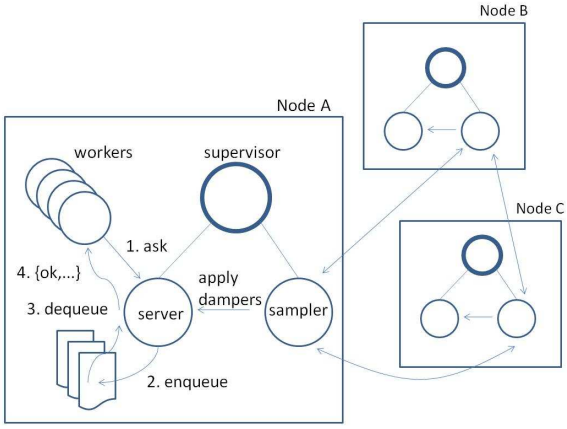

Figure 12. Architecture of the JOBS framework.

The job server selects a queue based on job type, and fetches the configured set of regulators for that queue. There are a number of different regulator types:

- `rate` – given a frequency *f*, ensures that the rate of accepted jobs does not exceed *f*.
- `counter` – ensures that the number of simultaneously executing jobs of the given type does not exceed the defined value.
- `group_rate` – the total rate of jobs accepted from all queues with the same group rate can not exceed the given frequency.

7.2.1 Counter-based regulation

Counter-based regulation provides a form of credit system. It can be used to reduce contention e.g. when running mnesia transactions, and has been found to actually improve throughput that way.

The counter system is based on gproc's counters and aggregated counters (see (Wiger 2007)), mainly to avoid reinventing the wheel. Each worker is assigned a counter with some given value. We make use of gproc's ability to handle complex aliases, and name the counter after the associated regulator, with a JOBS-specific prefix. Counters in gproc are "shared properties", so each process can have its own instance, which is exactly what we need. When the regulator is instantiated, an aggregated counter is also created. As new jobs are dispatched, the aggregated counter is automatically updated with the corresponding increment. This way, the aggregated counter provides the total value for regulation. If the worker process dies, gproc detects this and removes the counter, automatically adjusting the aggregated counter as well.

It would be possible for the dispatched process to access the counters, e.g. to reduce them incrementally for long-running jobs. No API for this has been implemented, but it could be done through the normal gproc counter API, and any changes would immediately be reflected in the aggregated counter.

7.2.2 Rate-based regulation

Rate-based regulation is done by remembering the time of the latest dispatch for each regulator. The number of jobs to dispatch at the time of a check is $\lfloor \frac{T_c - T_l}{I} \rfloor$, where T_c is the current time, T_l is the time of the last dispatch, and I is the pre-calculated dispatch interval corresponding to the configured maximum rate.

After each dispatch, if the queue is non-empty, a timer is set to the time remaining until the next dispatch, or 0, if the next dispatch is already overdue. If the queue is empty, it will not be checked again, until a new request arrives.

In other words, the rate regulator will always attempt to dispatch jobs at the maximum allowed rate. The dispatch rate will of course never be higher than the arrival rate.

7.2.3 Group rates

It is possible to group regulators, by specifying the `group_rate` option. When a regulator belongs to a group, the rate parameters of the group are updated each time a regulator is used. The least value from comparing the group regulator and the specific regulator is picked. This can be used to specify a maximum total rate of a group of requests, while allowing for greater peaks in the individual request types.

7.2.4 Feedback modifiers

Distributed feedback-based regulation is accomplished by letting the sampler processes exchange status information. The job server process receives an instruction to apply "modifiers", each indicating a specific sampler type and giving a type-specific "degree" of overload (a degree of 0 means no overload). Each regulator in each queue then determines individually how to respond to each damper, e.g. by reducing the nominal rate by some multiple of the degree.

7.3 Regulation API

The basic API for submitting to load regulation is:

```
jobs:ask(JobType) ->
    {ok, Opaque} | {error, Reason}
  Reason = rejected | timeout
```

If there is no registered job type matching `JobType`, an exception is raised.

The recommended way to end the job is to let the process terminate, i.e. create a temporary process for each job. If the process needs to be kept in order to perform more work, it must explicitly tell the job regulator when it has completed the job:

```
jobs:done(Opaque)
```

It is also possible to do this as a one-liner:

```
jobs:run(JobType, Fun) -> Result
```

Result is the result of calling Fun(), when the job request has been accepted. If the job request is denied with `{error,Reason}`, `erlang:error(Reason)` is raised.

7.4 Management API

It is possible to dynamically add, remove and reconfigure queues. This is partly intended for continuous maintenance and evolution of a system, but could also be used for dynamic regulation, e.g. changing the regulation parameters based on policy control, sampling feedback or operator intervention.

One could also imagine implementing a "training" function, where the feedback from the sampler is used to find an optimal throughput level. No experiments have been done with this yet.

7.5 Sampler Plugins

As the indicators for determining overload can vary significantly between systems, the sampler behaviour provides a plugin API. The callback API is designed to be as simple as possible:

```
%% Initialize the plugin;
%% called at startup or when plugin is added
init(Argument) ->
   {ok, InternalState}.

%% A message (e.g. a mnesia event) has arrived
%% to the sampler
handle_msg(Msg, Time, State) ->
   {log, Value, NewState} | {ignore, NewState}.

%% A sample interval has triggered.
%% Sample and return the result.
sample(Time, State) ->
   {Value, NewState}.

%% Calculate an overload factor, based on the history
%% of samples and possible indicators based on incoming
%% messages.
calc(History, State) ->
   {Factor, NewState}.
```

The history is simply a list of `{Time, Value}` tuples. A default function is provided for assessing the contents of the list:

```
calc(Type, Template, History) -> Factor :: term().
  Type     = time | value
  Template = [{Threshold, Factor}]
```

For example, a cpu load sampler plugin might provide a template like `[{80, 1}, {90, 2}, {100, 3}]`, meaning that the `Factor` is set to 1 at 80% overload, 2 at 90%, etc.

When e.g. sampling mnesia, the only information we can extract is whether it is overloaded or not – not the degree of load. It is then better to check the duration of the overload condition, e.g. with a template like `[{0,1}, {30,2}, {45,3}, {60,4}]`. If there is currently no overload, we do not even check the template. If there is overload (`Value == true`), we check how long it has been true, and find the lowest corresponding threshold in the template. In this case, if overload has persisted for 35 seconds, `Factor = 2`.

These values need to be tuned for the system in question.

(Some administrative functions, such as terminate() and code_change() will probably be added as well, in line with standard OTP behaviours.)

7.6 evaluation

It is important to note that, at the time of writing, JOBS has not yet been used in commercial operation. Promising results from prototypes have earned it a place in products currently under development.

In the prototype tests, we have used JOBS to regulate a system that uses a distributed, persistent mnesia database. The system tends to be disk-bound, and without load regulation, it risks triggering mnesia overload by outrunning the transaction log dumper (see 2.7). If the load persists, eventually the mnesia transaction manager (`mnesia_tm`) will also fall behind, building up a very large message queue. As each new application-level request is handled in a separate process, eventually enough processes will be backed up waiting for mnesia that the node crashes, either running out of memory, or running out of ETS tables (each mnesia transaction creates an ETS table for the temporary transaction store).

We regulated the system by adding a counter-based queue and empirically adjusting the number of allowed concurrent requests. Just by doing this, we increased the request rate within which we were able to meet the response time requirements. Our assumption is that reduced low-level contention is the main reason for this (see 2.5).

At some point, however, we still observed mnesia overload and subsequent eventual node crashes. We addressed this by adding a mnesia load sampler, and configuring the request queue to reduce the number of concurrent requests while mnesia overload was observed. This improved the situation, but as we were testing on different hardware (and virtual instances) with radically different disk throughput, we observed that the optimal throughput level and necessary reduction seemed to differ between the systems. We then

extended JOBS to allow the samplers to detect *persisting* overload, and correspondingly increase the factor by which the regulators should reduce throughput in order to cope.

A unit test exists that exercises a number of different scenarios. An interesting finding was that when running a sequential loop, calling an empty function via the JOBS framework (in essence measuring the overhead of the framework), the maximum sustained rate on a budget dual-core laptop was 500 requests/s. Using parallel evaluation instead, starting all requests asynchronously, a batch of 500 parallel jobs finished in 110 ms if the maximum rate was set to 5000 requests/s. This seems to be the maximum theoretical rate given the type of hardware, giving an amortized per-request overhead of the JOBS framework of less than 200 μs, before optimizations.

8. Conclusion

We have designed a generic load regulation framework based on experience from several Erlang-based products designed to withstand significant levels of overload. The basic principle is one of classifying and queueing jobs at the input edges of the system, in a manner similar to that used by the DiffServ mechanism for achieving quality of service in IP networks.

The idea of performing admission control at the input edges of the system is mainly inspired by the AXD 301 system and its derivatives, but it could just as well have been borrowed from the SEDA framework. JOBS is a more general implementation than the AXD 301-based load regulation, with a runtime management API and a plugin approach to queue management and feedback control.

We have found this to be a strategy that suits Erlang very well. We have tested the framework in a prototype for a commercial system, and found that the feedback mechanism works, even in a distributed system. Thus, it is possible to configure the framework to detect choke points and reduce the accepted traffic, as well as reduce it even more if the overload condition persists.

Possible future developments are development of more sampler plugins, a "training facility" which automatically finds an appropriate throughput level, and possibly also a parameterized queue type capable of performing load-balancing.

Another thought is that with a queue configured to invoke a certain function, we could have "producer queues", and essentially an adaptive and scriptable load generation tool.

The code for JOBS has been released as Open Source (ESL 2010).

Acknowledgments

I wish to thank the anonymous reviewer who brought the work on SEDA (Welsh et al. 2001) to my attention, and Kurt Jonsson (Ericsson, retired) for his outstanding work on performance profiling and load regulation in Erlang.

References

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM. doi: http://doi.acm.org/10.1145/1465482.1465560.

J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

G. Bell. Failure to thrive: Qos and the culture of operational networking. In *RIPQoS '03: Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*, pages 115–120, New York, NY, USA, 2003. ACM. ISBN 1-58113-748-6. doi: http://doi.acm.org/10.1145/944592.944595.

Ericsson. Erlang/otp efficiency guide, 2010a. URL `http://erlang.org/doc/efficiency_guide/users_guide.html`.

Ericsson. Erlang/otp overload reference manual, 2010b. URL `http://erlang.org/doc/man/overload.html`.

ESL. Jobs github repository, 2010. URL `http://github.com/esl/jobs`.

T. Gray. Why not atm?, 2000. URL `http://staff.washington.edu/gray/papers/whynotatm.html`.

S. Haver. Freak wave event at draupner jacket januar 1 1995, May 2003. URL `http://folk.uio.no/karstent/seminarV05/Haver2004.pdf`.

T. Kitaguchi and H. Hamada. Telecommunications service continuity and disaster recovery, 2008. URL `http://www.ieee-cqr.org/2008/Day%202/Session%208%20-%20Expert%20Panel/5%20Takaya%20Kitaguchi.pdf`.

F. L. Lewis. *Applied Optimal Control and Estimation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1992. ISBN 013040361X.

J. H. Nyström, P. W. Trinder, and D. J. King. High-level distribution for the rapid production of robust telecoms software: Comparing c++ and erlang, 2008.

M. Peuhkuri. Ip quality of service, May 2010. URL `http://www.netlab.tkk.fi/~puhuri/htyo/Tik-110.551/iwork.ps`.

P. H. Taylor, T. A. Adcock, A. G. Borthwick, D. A. Walker, and Y. Yao. The nature of the draupner giant wave of 1st january 1995 and the associated sea-state, and how to estimate directional spreading from an eulerian surface elevation time history. In *9th International Workshop on Wave Hindcasting and Forecasting*, 2006.

Theorem.net. On-line introductions to control theory and engineering, 2001. URL `http://www.theorem.net/theorem/background.html`.

Y. Wang. Bassoon: Backpressure-based sip overload control, 2010. URL `http://research.csc.ncsu.edu/netsrv/?q=bassoon`.

M. Welsh and D. Culler. Virtualization considered harmful: Os design directions for well-conditioned services. *Hot Topics in Operating Systems, Workshop on*, 0:0139, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/HOTOS.2001.990074.

M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A design framework for highly concurrent systems, 2000.

M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/502059.502057.

U. Wiger. Four-fold increase in productivity and quality - industrial-strength functional programming in telecom-class products, 2001.

U. Wiger. Extended process registry for Erlang. In *ERLANG '07: Proc. of the 2007 SIGPLAN workshop on ERLANG Workshop*, pages 1–10, New York, NY, USA, 2007. ACM.

Wikipedia. Poisson process, 2010. URL `http://en.wikipedia.org/wiki/Poisson_process`.

B. Wilson. Reddit case study, May 2008. URL `http://www.ski-epic.com/2008_reddit_case_study/index.html`.