Autonomic Management of Client Concurrency in a Distributed Storage Service

Markus Tauber, Graham Kirby and Alan Dearle
School of Computer Science, University of St Andrews,
North Haugh, St Andrews KY16 9SX, Scotland
Email: {markus, graham, al}@cs.st-andrews.ac.uk

***Abstract*—A distributed autonomic system adapts its constituent components to a changing environment. This paper reports on the application of autonomic management to a distributed storage service. We developed a simple analytic model which suggested potential benefit from tuning the degree of concurrency used in data retrieval operations, to suit dynamic conditions. We then validated this experimentally by developing an autonomic manager to control the degree of concurrency. We compared the resulting data retrieval performance with non-autonomic versions, using various combinations of network capacity, membership churn and workload patterns. Overall, autonomic management yielded improved retrieval performance. It also produced a distinct but not significant increase in network usage relative to one non-autonomic configuration, and a significant reduction relative to another.**

***Keywords*-autonomic management; distributed storage**

I. Introduction

We are interested in distributed storage services that harness surplus storage capacity. The perceived performance of such services depends on their configuration parameters and on various dynamic conditions. For given conditions, one configuration may be better than another, with respect to measures such as resource consumption and performance.

For our investigation we used a simple replicated storage service [1], in which a client performing a read operation may retrieve data from any of a number of identical replicas stored on various servers. The client may decide which replica to attempt to retrieve first, and also whether to retrieve replicas sequentially or in parallel.

This sequential/parallel retrieval behaviour is controlled by the client's *degree of concurrency* configuration parameter (C). When C is set to 1, replica retrieval attempts are made sequentially, continuing until a replica is successfully retrieved. At the other extreme, when C is set to the number of replicas, all retrieval attempts are initiated concurrently, terminating when the first successful result is returned.

The optimal C value depends on dynamic conditions, giving potential scope for autonomic management [2]. A high value is desirable when there is significant unpredictable variability in the times taken to retrieve individual replicas, or when servers exhibit a high failure rate. In such cases a parallel retrieval strategy is likely to return a result more quickly than a sequential strategy.

Conversely, a low C value is desirable when there is low variability in retrieval time and there is a network bottleneck close to the client. In this situation the low variability removes any performance benefit of parallel retrieval, while the effect of the bottleneck is that parallel retrieval would increase retrieval time due to contention.

We developed an analytic model to investigate the potential of autonomic management in this context. As it indicated promising results, we then implemented a simple autonomic manager to control the C parameter depending on monitored conditions. We experimentally evaluated the effects of the manager on data retrieval time as perceived by the user, and on network usage. The experiments were conducted using a deployed storage service, subjected to various membership churn, workload and network speed patterns.

The autonomic manager successfully detected and corrected situations in which the C parameter was set inappropriately, without any prior knowledge of the network conditions or workload, yielding an improvement in performance when compared with statically configured clients. Network usage was slightly increased compared to one non-autonomic configuration, and reduced relative to another.

II. Related Work

Distributed storage systems adopt various strategies to attempt to optimize performance. In *PAST* [3], the underlying peer-to-peer overlay *Pastry* [4] transparently prioritizes servers with good performance during routing operations. This means that Pastry first routes to servers with good performance when a data item is requested.

In *CFS* [5], *Ivy* [6] and *GFS* [7] a client determines, for an individual request, the server from which it fetches a replica based on a performance measure computed by some *Server Ranking Mechanism* (SRM). The objective of such a SRM is to improve data retrieval performance by ranking servers based on predictions about which host will result in the shortest retrieval time. This is based on the assumption that historical monitoring data can be used to predict future performance of specific hosts.

We do not know of any other work using dynamic control of the degree of concurrency.

III. Effects of Concurrent Retrieval

In this section we develop a simple analytical model to demonstrate the effect of the C parameter in various situations. We assume a distributed storage system with the following properties:

- For a given data item, a client knows the addresses of up to *R* different servers storing identical copies.

- Replicas can be individually verified, thus only one replica need be retrieved successfully.
- If a replica cannot be retrieved, the client attempts to retrieve one from a different server.

The model calculates the overall time to complete a user retrieval request; we use it to compare the effects of low and high C values. We also investigate the effect of availability of an SRM oracle that is able to perfectly rank a set of servers with respect to the time needed to retrieve a replica from each one. This gives insight into the potential benefits of a practical SRM.

A. Analytical Model

The analytic model is based on a simplified distributed storage service comprising a single client communicating with N servers. Each data item is replicated on R servers. The client and servers are connected via an interconnection, whose internal network links are assumed to exhibit significantly higher bandwidth and lower latency than the links between participants and the interconnection. Thus, the time to transfer data across the interconnection is assumed to be negligible.

Each user-level read request is serviced by a *get* operation executed on the storage client. This results in one or more *fetch* operations to retrieve replicas from specific servers.

The parameters of the model are as follows:

- R: the replication factor
- S: the average data item size
- C: the degree of concurrency, constrained to either 1 or R for simplicity
- P: the probability of failure of a *fetch* operation
- F: the average time for the client to detect failure of a *fetch* operation
- B_i: the perceived bandwidth between participant *i* and the interconnection
- L_i: the perceived latency between participant *i* and the interconnection

The expected *get* time is influenced by these parameters and by the availability or otherwise of an SRM oracle. When C is low, yielding largely sequential replica fetches, the number of fetches increases with P, as does the resulting *get* time. An SRM oracle is only useful with a low C value. By definition the oracle always chooses a non-failing server, thus the *get* time is governed by the lowest *fetch* time.

The lowest *fetch* time is also the most significant factor when C is high, since the *get* operation completes when the first *fetch* operation completes successfully. Additionally, when the client link to the interconnection is a bottleneck, the *fetch* and *get* times increase with C due to contention between concurrent retrievals.

1) Fetch Time: We first derive the *fetch* times for individual replicas in terms of the model parameters. The *fetch* time for replica i has three components:

- $t_{request\_server\_i}$: the time for the request to reach server i from the client
- $t_{response\_server\_i\_link}$: the time for the replica data to reach the interconnection from server i
- $t_{response\_client\_link}$: the time for the replica data to reach the client from the interconnection

The size of a request message is negligible, so only latencies are significant:

$$t_{request\_server\_i} = L_{client} + L_{server\_i} \tag{1}$$

Time $t_{response\_server\_i\_link}$ is determined by the replica size and the bandwidth and latency of the server link:

$$t_{response\_server\_i\_link} = \frac{S}{B_{server\_i}} + L_{server\_i} \tag{2}$$

The last component is calculated similarly, but the available bandwidth is shared among C concurrent replica transfers:

$$t_{response\_client\_link} = \frac{S}{\left(\frac{B_{client}}{C}\right)} + L_{client} \tag{3}$$

The overall *fetch* time for server i is:

$$\begin{aligned} t_{fetch\_i} &= 2L_{client} + 2L_{server\_i} + \\ & S\left(\frac{1}{B_{server\_i}} + \frac{C}{B_{client}}\right) \end{aligned} \tag{4}$$

Observe that a high C value always raises individual *fetch* times relative to a low C value, but that the effect becomes less significant as the client bandwidth increases.

2) Get Time: The significant components of the overall *get* time differ, depending on the C value and whether or not an SRM oracle is used. The sensible configurations are:

case	description	C	SRM oracle
1	low concurrency, no SRM	1	no
2	low concurrency, with SRM	1	yes
3	high concurrency, no SRM	R	no

In case 1, the *get* operation starts by fetching a replica from a randomly selected server, completing if the replica is retrieved successfully. If the *fetch* operation fails, with probability P, this is detected after time F, and a different replica is tried. The overall *get* operation fails if all individual *fetch* operations fail, with probability P^R.

The expected total time for a successful *get* operation is the sum of the average fetch time and the time involved in dealing with any failures. The number of failures, k, lies between 0 and $R-1$. For a given k, the time involved is kF, while the probability of that number of failures occurring is P^k. Hence, overall, the expected time to deal with failures is the weighted sum for all values of k:

$$t_{get\_case\_1} = t_{fetch\_avg} + \sum_{k=0}^{R-1}(kFP^k) \tag{5}$$

In case 2, low concurrency with SRM, the *get* operation initiates a single *fetch* operation from the best server as determined by the oracle. Since its predictions are perfect, this operation succeeds if any non-failed servers exist. Thus the individual *fetch* time, as modelled in formula 4, determines the overall *get* time.

In case 3, high concurrency with no SRM, the *get* operation initiates C concurrent *fetch* operations. Again, the fastest successful *fetch* determines the overall *get* time.

$$t_{get\_cases\_2\_and\_3} = t_{fetch\_min} \quad (6)$$

Observe that a high C value (case 3) eliminates the influence of the failure detection time F, but, as noted earlier, raises the overall time when there is a client bottleneck.

A low C used with an SRM oracle is the optimum combination, since failures do not contribute to the overall time, nor are individual *fetch* times increased by client link contention. Unfortunately a perfect SRM is not realizable. However, even an imperfect SRM that makes better predictions than chance may be beneficial, in that using its predictions to decide *fetch* order for low C may prioritize servers with better than average response times.

From this, we hypothesize that a good client strategy is to set C to a low value when there is a client bottleneck, or there is little variability in response times among servers. The rationale for the latter is that if all concurrently fetched replicas take a similar time, there is little benefit gained by being able to complete after the first is received. Furthermore, replicas should be tried in the order recommended by an SRM that monitors server response times.

Conversely, C should be set to a high value in situations when there is no client bottleneck and there is high, unpredictable, variability among servers. In this case an SRM will be ineffective, and the best server response time will be significantly lower than the average.

Given that the optimum value of C depends on the location of the bottleneck, if any, and the variability in server response times, dynamic adaptation of C offers several potential benefits. Firstly, the bottleneck location may not be not known statically, or may change dynamically due to client mobility. Even if it were known statically, a requirement for manual configuration of C may be undesirable. Secondly, dynamic adaptation allows response to unpredictable variation in server response times, perhaps due to fluctuations in server or network load.

While this model is simplistic, it has served its purpose in providing sufficient indication of the potential benefits of dynamic C adaptation to justify implementing and evaluating an autonomic manager and SRM.

B. Example Scenarios

We illustrate the model by plotting the predicted *get* times for selected parameter values, showing the effect of various *fetch* failure probabilities for the three configuration cases. Failure of fetch operations may be caused by faulty servers or by churn in the server population.

We fix the following parameter values:

- R: 4 (replication factor)
- S: $500KB$ (average data item size)
- F: $2s$ (average time to detect failure)
- L_i: $100ms$ for client and all servers (latency)

Figure 1 shows the model's predictions for a client-side bottleneck with the following bandwidth values:

- B_{server_i}: $10MB/s$ for all servers
- B_{client}: $0.1MB/s$

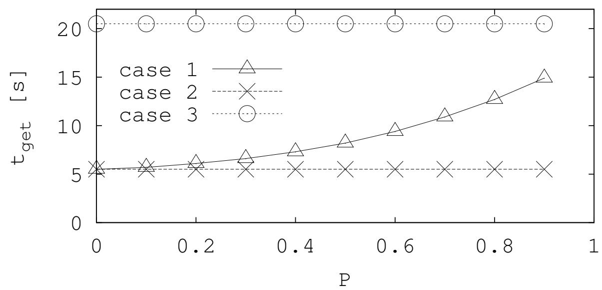


Figure 1: Effect of *fetch* failure probability on *get* time, with client-side bottleneck.

This shows that low C is desirable at all failure rates in this case. The benefit of the SRM oracle in being able to predict a non-failing server becomes more significant as failure rates increase.

Figure 2 shows the model's predictions for a server-side bottleneck with the following parameter values:

- B_{server_i}: $0.1MB/s$ for all servers
- B_{client}: $10MB/s$

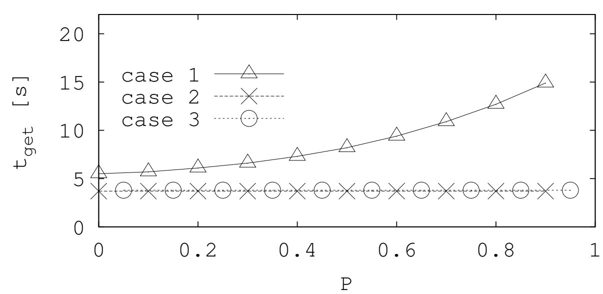


Figure 2: Effect of *fetch* failure probability on *get* time, with server-side bottleneck.

This shows that all three client configurations give similar results for low failure rates, with high C or an SRM oracle giving better results as failure rates increase.

IV. Autonomic Management of Concurrency

An autonomic manager to control C was implemented using the Generic Autonomic Management Framework (GAMF) [8], [9], structured around a monitor/analyze/plan/execute cycle [2].

An instance of the manager running the control cycle was installed on each client, while an instance of the SRM was installed on the client and on each server. Details of the manager's control cycle are given in the following subsections.

A. Monitoring

Latencies and bandwidths were monitored by the SRM instances, using periodic pings with various packet sizes.

The following quantities were monitored by the manager:

- perceived latencies between the client, interconnection and servers, as reported by the SRM
- perceived bandwidth between the client, interconnection and servers, as reported by the SRM
- rate of initiated fetch operations
- rate of failed fetch operations

B. Analysis

The following metrics were derived from the monitored data:

- FFR: the *fetch failure ratio*, comprising the ratio of recent failed to initiated *fetch* operations, or 0 if no operations were initiated
- EFT: the *expected fetch time* for each server, estimated from recent measured latencies and bandwidths
- FTV: the *fetch time variation* between servers, comprising the ratio of the standard deviation in the most recent EFT values, to the mean
- BN: the *bottleneck* value, comprising the ratio of recent monitored client-interconnection bandwidth to the mean of server-interconnection bandwidths

It was decided to base FTV on estimated rather than actual observed *fetch* times, since this allowed values to be generated independently of workload. Note that the metric considered only recent variation between servers; it did not take account of variation in the performance of individual servers over time.

C. Planning

During each iteration of the autonomic cycle, the manager used the generated metrics to decide the next value of C, which was allowed to take any integer value from 1 to R[1]. Changes to C were triggered when observed metric values crossed certain *high* and *low* thresholds for each metric, defined by statically configured parameters of the autonomic manager. The manager's policy was defined as follows:

if $C < R$ **then**
  if $FFR\ is\ high$ **and** $FTV\ is\ high$ **then**
    $C \leftarrow R$
  else if $FFR\ is\ high$ **or** $FTV\ is\ high$ **then**
    $C \leftarrow C + 1$
  end if
else if $FFR\ is\ low$ **and** $FTV\ is\ low$ **then**
  if $BN\ is\ low$ **then**
    $C \leftarrow 1$
  else
    $C \leftarrow C - 1$
  end if
end if

[1]Note that this is less restrictive than the analytic model.

The policy was designed to increase C in situations when there was a high server failure rate or a high variation between server response times, and to do so more aggressively when both of these conditions held. Conversely, the policy reduced C when the failure rate and variation were both low, more aggressively if there was a client-side bottleneck.

D. Execution

The execution phase involved simply setting the C parameter. The client *get* algorithm was also adapted to incorporate advice from the SRM, so that whenever C was set at less than R (i.e. not all *fetch* operations were issued in parallel), higher ranking servers were prioritized. The SRM did not attempt to predict probability of server failure, but considered only recent history of server connectivity by selecting the server with the lowest recent EFT value.

V. Experimental Evaluation

Three configurations of the autonomically managed client, using different values for the FFR threshold parameter, were evaluated against two non-managed clients using fixed low and high C values respectively. The effects of the various policies on performance and resource consumption were measured in a local-area storage service deployment, exposed to various patterns of data item size, workload, server churn and network conditions.

In each experiment, 16 storage servers were deployed in an isolated test-bed. Traffic was routed through a traffic shaper, allowing various network conditions to be simulated.

A. Experimental Parameters

The following data item sizes were used:

- $0.1MB$
- $1MB$

The following workload patterns were used:

- *heavy-weight*, comprising 300 sequential *get* operations
- *light-weight*, comprising 10 sequential *get* operations, with $120s$ delay between each successive operation
- *variable-weight*, comprising 10 repetitions of a pattern containing 3 sequential *get* operations followed by $120s$ delay

The following churn patterns were used:

- *none*, in which all servers remained available
- *high*, in which each server alternated between on-line phases of about $40s$ and off-line phases of about $30s$

- *temporally varying*, in which the servers alternated between no- and high-churn phases lasting about $5min$

Higher churn rates led to higher fetch failure rates, since servers were more likely to be unavailable when required.

The following network patterns were used:

- *static with client bottleneck*, in which network conditions remained constant throughout, with greater connectivity for servers than client: server bandwidth $2.8MB/s$, server latency 0, client bandwidth $0.4MB/s$, client latency $20ms$
- *static with server bottleneck*, in which network conditions remained constant throughout, with greater connectivity for client than servers: server bandwidth $0.4MB/s$, server latency $20ms$, client bandwidth $9.8MB/s$, client latency 0
- *temporally varying*, in which the bandwidths and latencies of the various links randomly varied every $10s$

The shaped bandwidth figures were chosen to fit within the available physical bandwidth.

B. Management Policies

The configurations of the management policies are shown in table I. Policies 1 and 2 involved statically configured C values with no autonomic management. Policies 3 to 5 all used the same threshold values for the FTV and BN metrics, differing only in the FFR thresholds. This meant that the autonomic policies varied in the *fetch* failure rates required to trigger action.

policy	T_{FFR}	T_{FTV}	T_{BN}	initial C
1	-	-	-	1
2	-	-	-	4
3	0.1	0.2	0.8	1
4	0.3	0.2	0.8	1
5	0.5	0.2	0.8	1

Table I: Management policy configurations.

For policies 3 to 5 the frequency of the autonomic cycle was set to once per minute. Network latency and bandwidth observations were made by the SRM every $15s$.

C. Results

A series of experiments was performed, evaluating all combinations of data item size, workload, server churn and network conditions. Each experiment was repeated three times. Due to space constraints we highlight the most significant observations here; full results are reported in [1].

Figure 3 shows observed *get* throughput for the various policies, averaged over all experiments[2]. It can be seen that all autonomic policies yielded similar results, with a small but significant improvement over the static low-concurrency configuration (policy 1), and a greater improvement relative to the static high-concurrency configuration (policy 2).

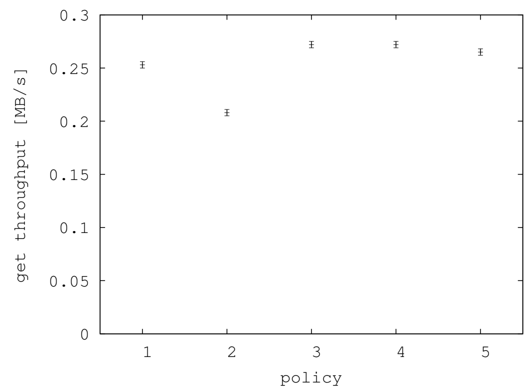


Figure 3: *Get* throughput averaged over all experiments.

Figure 4 shows the observed *network usage* figures, averaged over all experiments. Again there was little to distinguish between the autonomic policies. They yielded a distinct but not significant increase in network usage relative to policy 1, and a significant reduction relative to policy 2.

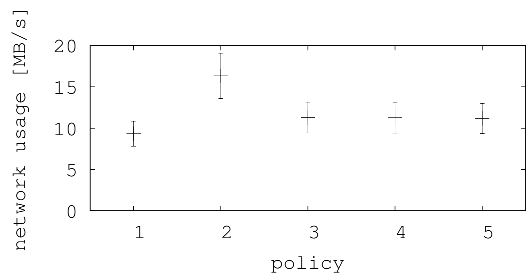


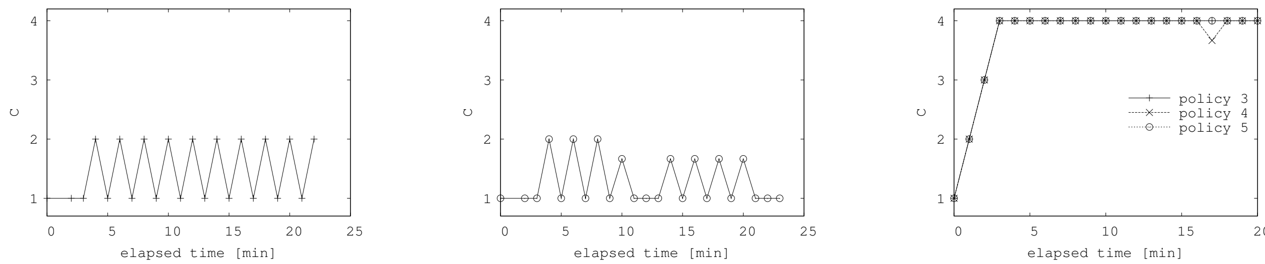
Figure 4: Network usage averaged over all experiments.

Figures 5a and 5b illustrate the effects of autonomic management in one example experiment, showing the progression of C values over time. This experiment involved large data items, high churn, heavy-weight workload and client-side bottleneck. The effects of policy 4 are omitted since they were almost identical to those of policy 3.

In this case low C gave the best *get* throughput. This was due to the SRM being able to detect good servers since network conditions remained static, while the client-side bottleneck caused network contention for higher C values. It can be seen in the plots that the autonomic managers successfully deduced that C could be kept at a low level and maintained it close to the optimum. As expected, policy 5, which used the highest threshold to decide when FFR was *low*, set the lowest C values.[3] Non-integer values for C

[2]Throughput is reported here, rather than specific *get* times, since multiple data item sizes are included.

[3]At first glance it appears that all the managers were over-eager to adjust C, given the saw-tooth patterns. However, it should be remembered that changing C does not incur any significant cost. This might be more serious in other autonomic schemes—for example, if the parameter being tuned controlled the physical location of data.



(a) Policy 3, client-side bottleneck.

(b) Policy 5, client-side bottleneck.

(c) Policies 3-5, temporally varying network conditions.

Figure 5: C progressions for various network conditions.

occur due to averaging of multiple runs.

Figure 5c shows an experiment with large data items, low churn, light-weight workload and temporally varying network conditions. High C gave better results, since the varying network conditions hampered the SRM's ability to predict server performance. It can be seen that the autonomic managers detected that C could be kept at a high level.

VI. Conclusions

We have demonstrated that autonomic management of the degree of replica retrieval concurrency in a distributed storage client can achieve a small but significant improvement in performance at a small cost of additional resource consumption. Under changing conditions, autonomic management can adapt concurrency to suit prevailing conditions, while under constant conditions that are not statically known, it can converge to an appropriate value.

By definition, autonomic management of C is only of benefit when there is no statically known fixed C that gives optimum results. The observed benefits are thus closely dependent on the chosen experimental conditions.

We discovered, through conducting the experiments, that a low C fixed configuration was better than a high C fixed configuration for the majority of the conditions tested. This is why policy 1 shows better throughput than policy 2 in figure 3. The autonomic policies perform better still, because they are able, effectively, to select either policy 1 or 2 automatically. However, the improvement over policy 1 is not large, because there were not many experiments for which policy 2 was better. We therefore hypothesize that autonomic management would exhibit greater benefit in a series of experiments with a more even balance between conditions favouring policy 1 and conditions favouring policy 2.

This ability to auto-select a value for C is a strong feature of autonomic management. Another is its ability to dynamically adjust C to accommodate dynamically changing circumstances. We would expect to see greater benefits from autonomic management with experiments specifically designed to require periodic change. These might include conditions in which a network exhibits alternating phases of predictable and random behaviour.

References

[1] M. Tauber, "Autonomic Management in a Distributed Storage System," Ph.D. dissertation, University of St Andrews, School of Computer Science, 2010, arXiv:1007.0328v1.

[2] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[3] P. Druschel and A. Rowstron, "PAST: A Large-scale, Persistent Peer-to-peer Storage Utility," in *HotOS VIII*, May 2001.

[4] A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001, pp. 329–350.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area Cooperative Storage with CFS," in *SOSP '01: 18th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM Press, 2001, pp. 202–215.

[6] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A Read/Write Peer-to-peer File System," in *The Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[7] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," in *SOSP '03: 19th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM Press, October 2003, pp. 29–43.

[8] M. Tauber, "A Generic Autonomic Management Framework (GAMF)," February 2010. [Online]. Available: http://www-systems.cs.st-andrews.ac.uk/gamf

[9] M. Tauber, G. Kirby, and A. Dearle, "Self-Adaptation Applied to Peer-Set Maintenance in Chord via a Generic Autonomic Management Framework," in *SAN: Self-Adaptive Networks Workshop, IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO2010)*.