The Dynamic Enterprise Bus[*]

Dag Johansen
Dept. of Computer Science
University of Tromsø, Norway

Håvard Johansen
Dept. of Computer Science
University of Tromsø, Norway

Abstract

In this paper we present a prototype enterprise information run-time heavily inspired by the fundamental principles of autonomic computing. Through self-configuration, self-optimization, and self-healing techniques, this run-time targets next-generation extreme scale information-access systems. We demonstrate these concepts in a processing cluster that allocates resources dynamically upon demand.

1 Introduction

Enterprise search systems are ripe for change. In their infancy, these systems were primarily used for information retrieval purposes, with main data sources being internal corporate archives. The service they provided resembled much traditional Internet search services. Emerging trends in enterprise computing, though, is to use search technology as an embedded part of business-critical applications like, for instance, business intelligence, e-commerce, surveillance, security, and fraud detection. Consequently, traditional search software needs to change to meet new requirements. We denote the next generation of enterprise search software as *information access* technology, capturing the notion that the software resides between large heterogeneous data repositories and complex data intensive applications. This is where databases and enterprise run-time software used to play key roles.

Traditional Internet search services are typically provided by complex networks of up to several hundred thousands of interconnected computers. Configuring and maintaining these large-scale clusters is a daunting task. Hence, only a handful companies have sufficient resources and skills to provide such services. At the same time, many corporations are in need of this type of technology in-house, but do not have the necessary manpower, economy, and technical skill to even consider such solutions.

[*]This work is supported in part by the Research Council of Norway through the National Center for Research-based Innovation program. Dag Johansen is also chief scientist at Fast Search & Transfer.

This is where autonomic computing [18] comes into play. Our goal is that information access software should keep manual control and interventions outside the computational loop as much as possible. Closely related is that such autonomic solutions should consolidate and utilize resources better, with the net effect that large-scale information access systems can be built in smaller scale. In this vein, self-configuration, self-optimization, and self-healing become important.

We are building the next generation run-time for future generation information access systems. Autonomic behavior is fundamental in order to dynamically adapt and reconfigure to accommodate changing situations. Hence, the run-time needs efficient mechanisms for monitoring and controlling applications and their resource, moving applications transparently while in execution, scheduling resources in accordance to end-to-end *service-level agreements* (SLAs), adding functional replicas to handle sudden load spikes, and the like.

In this paper, we present a prototype run-time with self-configuration, self-optimization, and self-healing properties. Key to support these features are accurate and efficient monitoring and control components. The paper is organized as follows. First, we present the overall architecture and high-level aspects of our run-time. Then, in Section 3, we present a prototype implementation with relevant experiments. In Section 4, we discuss aspects of our work, with related work presented in Section 5. Finally, Section 6 concludes.

2 The Dynamic Enterprise Bus

We denote our run-time the *Dynamic Enterprise Bus* (DEB). DEB incorporates autonomous behavior and has the following properties. First, DEB is *self-configuring*. Installing and configuring enterprise systems have fundamental problems often ignored in academia, but that are highly realistic when a front-line industry engineer is on a customer location deploying complex search software integrating with a terabyte data store. At deployment, DEB supports a worm-like growth of the computation, much the way

0-7695-3093-1/08 $25.00 © 2008 IEEE
DOI 10.1109/ICAS.2008.43

worm applications grew on demand [15] or itinerant-style mobile agents were applied [9]. As such, an information-access application can be deployed initially on one or a few physical machines, and grow dynamically into a larger physical deployment while warming up with real data.

Second, DEB provides *self-optimization* properties by integrating distributed fine-grained monitoring with a reactive control scheme. Traditionally, this type of monitoring-control loop, often defined as a reactive system, has been applied in robotics and embedded systems. However, we conjecture that this type of autonomic behavior has applicability in management of, for instance, large-scale enterprise and Internet applications serving millions of interactive users, like e-commerce and enterprise search platforms, real-time information filtering applications in the financial sector, and enterprise business intelligence systems. These are applications with extreme availability requirements.

By enabling DEB to autonomously make policy decisions based on collected real-time metrics, applications can be dynamically adjusted to use different availability, reliability, and consistency tradeoffs in accordance to measured real-time load. For instance, during load spikes, a weaker consistency model might be used to ensure high availability. Also, if multiple applications run concurrently, resource allocations can change in accordance with per application load and priority.

To facilitate this type of policy decisions, our monitoring scheme collects run-time metrics from the whole software stack including data from the hardware level, the virtual-machine level, and data from the individual application components. Using this data, a DEB computation can be dynamically adjusted to, for instance, the underlying physical platform and grow and shrink on demand.

Accurate failure-detection is instrumental to enable our third autonomic property, *self-healing*. Since we have integrated monitoring throughout DEB, we attempt to efficiently detect failures and recover from them in a transparent manner. That is, once an application component is detected as failed, recovery actions might, for instance, have other components compensate for that failure.

2.1 Computational Model

Based on our experience with existing enterprise search systems, we have devised a computational model or structural pattern that often occurs. It is a computational model where data flows uni-directionally through a sequence of computational stages, with output from one stage used as input to the next stage. This structure is typically found in existing search systems in the document processing pipeline residing between a document crawler and the indexing software. From a few to almost a hundred different functional components can be found in such a pipeline. The query pro-

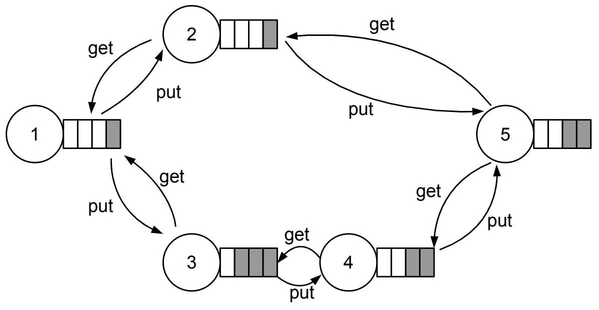

Figure 1. DEB processing pipeline with five stages

cessing pipeline is similarly structured, where user queries are sent through a series of computational stages before being matched against an index store.

In our model, each pipeline stage is a processing element that communicates through a publish-consume abstraction. We use outbound queues, where multiple successor stages can subscribe to its content and pull (consume) elements from the queue on demand. This design decision is taken to ease scalability and fault tolerance. Inserting a stage is, for instance, done by bootstrapping it and then letting it start consuming elements from its predecessor stage. Hence, a computation is by default modelled as a directed acyclic graph. Figure 1 illustrates a DEB pipeline with five computational stages.

2.2 Application-Level Failure Detection

Failure detection is typically an integrated part of a low-level fault-tolerance mechanism. We propose to add failure detection to the application itself as a non-functional component. This is inspired by the end-to-end argument [13], a principle advocating that some functionality is better provided outside the communication substrate.

DEB behavior that qualify as failures are specified by the overlying applications through the insertion of application specific SLAs. Simply put, a component, system, or application is said to fail if it does not meet its guaranteed SLAs. We denote this as a *quality failure*.

Quality failures resemble hard real-time failing models [1], but is more extensive since SLAs can contain more than just meeting a specific deadline. One example is that end-to-end processing latency of a document can have a maximum bound limit. For instance, a customer integrating an enterprise search solution with a net auction service might have a requirement that new or updated data elements must be indexed within a certain number of milliseconds. If this quality level is not met in practice, the system has failed from the customer's perspective.

We also argue for failure handling being a first order design consideration. Fault-tolerance schemes are often retrofitted into existing architectures as an afterthought, but we suggest an explicit application-level failure handling mode of operandi. This is in particular true with our failure model, where standard replication schemes do not necessarily work. The state machine approach [14], for instance, works well for fail-stop and Byzantine failures, but its inherited replication scheme might add to the type of failures we attempt to solve. Meeting an end-to-end SLA may require use of as little redundant computing as possible. One exception is situations where enough resources can be added, as cleverly done in, for instance, the Google MapReduce computation [4] when a fan-out computation is lagging behind.

2.3 Stateful Failure Detection

We consider a failure detector as a fundamental building block for high-available autonomous systems. There are, however, a number of problems related to failure detection. One obvious problem is to correctly detect remote events, like, for instance, distinguishing a crashed machine from a slow machine or a transient network failure. Another, and important problem in an autonomous system, is false positives when failures are detected too eagerly.

Consider a situation where a false positive is disseminated. This can, for instance, trigger a self-healing action by the autonomous system. A commonly used technique for failure handling is to abort an action and redirect a request or job to another replica. This often implies extra overhead including state-transfer and process activation. Meanwhile, the presumably failed machine becomes available again and recovers activity, maybe after a transient network error. In this case, the recovery action was triggered too fast and the costly overhead of failure-handling could have been avoided.

As such, we make two contributions here. First, we introduce a *stateful failure detector* in DEB. In its simplest form, it can be a log of sent and received ping messages between two machines in the system, which can be used to check the current situation against a historic record. Maybe a potential failure is something that occurs regularly at 3:00 AM–3:05 AM due to a local cron job, which should be detected as a normal situation and not as a failure.

Second, we add control functionality to the DEB failure detector such that distributed management policies can be implemented. For instance, a proactive recovery scheme can be tied to a local failure detector, so that emerging trends and suspicions can trigger some local management action. This can be local garbage collection, preemption of running jobs, and the like.

3 The DEB Run-Time

Important components of our DEB run-time architecture include monitoring, control, failure detector, and pipeline processing functions. A DEB installation spans one or more machines, each machine hosts one or more DEB nodes. Each node contains core DEB functions like monitoring and control, but can be specialized to perform specific tasks like coordination and pipeline processing.

When starting, a DEB node announces its presence to the DEB run-time by sending it a register message that includes the node's unique name, network listening port, and current state. The DEB monitor collects and maintains historic records of such changes to the system's state, including OS and hardware level attributes like CPU and network load. These records are made available to all nodes. In our current prototype, we use the Ganglia cluster monitoring system [12] to both collect and distribute system metrics.

Applications register pipeline configurations and SLAs with the run-time. Each pipeline configuration contains the description of which computational steps that are to be performed and in what order they are to be executed. Each SLA describes constraints on the run-time attributes. Upon receiving the pipeline configuration and the SLA, DEB self-organizes in order to accommodate the application, which includes allocating one or more nodes to execute the pipeline processing stages and connecting the stages together in the correct order. To avoid inconsistencies, global changes like node allocations are coordinated by a *master control node*.

To illustrate the ability of DEB to self-configure, we constructed a synthetic CPU bound pipeline consisting of three sequential stages: a generator stage, a compute stage, and a fuse stage. The generator stage injects the data items that are to be processed. Each data item contains a monotonically increasing sequence number, the time of its creation, some random data, and a load factor. The compute stage executes a synthetic CPU intensive operation on each received data item. The number of CPU cycles required to complete one particular data item is linearly depending on the data item's load factor. The fuse stage collects all processed data items and reports end-to-end latency.

To execute this pipeline, we deployed it on an eight node DEB installation, $n_0, \ldots, n_7$, spanning eight machines. All eight machines were equipped with two quad-core Intel Xenon processors and 8 GB of RAM. Each second, each node reports its status, including the stage it is executing and to which other nodes it is connected. Initially, all nodes were idle with node n_0 acting as the master control node. Upon receiving the pipeline configuration, n_0 allocated node n_1 to run the generator stage, n_2 to run the compute stage, and n_3 to run the fuse stage. The remaining nodes $n_4, \ldots, n_7$ were idle. Next, the control node in-

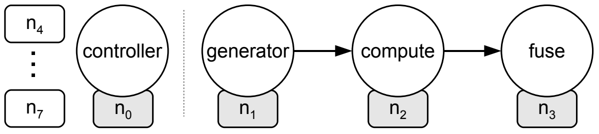

Figure 2. Initial test configuration

structed each node in turn, starting with n_3, to receive data items from the previous pipeline stage, resulting in the initial configuration shown in Figure 2.

Next we illustrate the DEB's ability to adapt the pipeline to changing load. We do this by adding a SLA that specifies an upper threshold value of 1.5 seconds on the completion of each data item. We then increase the processing load of the compute stages by increasing the data items load factor linearly with their sequence number. Figure 3 plots the resulting end-to-end latency for data items over time and the number of allocated compute stages. As can be seen from the figure, at time 335 seconds the end-to-end latency exceeds the set threshold of 1.5 seconds. At this time a quality failure is detected. The DEB failure handler then inspects the pipeline to identify the stage that adds most to the total end-to-end latency, which in this particular scenario will be the compute stage. To compensate, the failure handler increases the pipeline's processing capacity by adding a replica of the compute stage to the idle node n_4. Consequently, the end-to-end latency drops below the threshold. As the load continues to increase, similar quality failures are detected at times 725, 1069, and 1369. Each time, the failure handler compensates by deploying another compute stage to an idle node, resulting in an immediate latency drop.

To illustrate the DEB's ability to self-heal, we kept the load factor stable. We then crashed one of the nodes running a compute stage. As shown in Figure 4, the load factor is initially kept stable at the level where four nodes run the compute stage, resulting in a stable end-to-end latency below the threshold of 1.5 seconds. At time 559 seconds, when we crashed a node, the measured latency spiked due to the reduction in processing capacity. Next, the failure handler compensates by allocating another node to run the failed compute stage, which brought the latency back down.

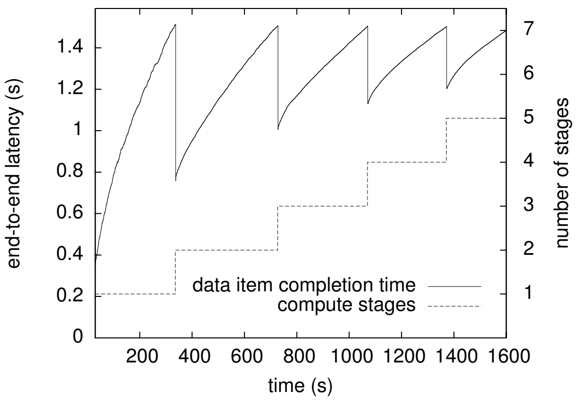

Figure 3. Adapting the processing pipeline to linearly increasing processing load

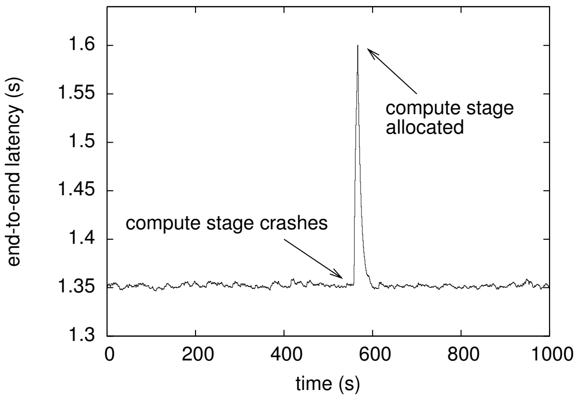

Figure 4. Adapting pipeline to a compute stage failure

4 Discussion

We have demonstrated our DEB run-time prototype, which evolves and adapts to meet SLAs. These SLAs have been negotiated prior to application deployment, and are formally agreed upon contracts between the DEB run-time and its applications. This service quality guarantee determines how the DEB run-time system performs self-configuration, self-optimization, and self-healing. This allows DEB to transparently compensate for failures and mitigate performance bottlenecks by, for instance, adding pipeline stages as demonstrated in Section 3.

Accurate failure detection is a pivotal component in DEB. We therefore monitor and collect a wide-range of run-time metrics, which are used as input to the DEB failure detection mechanism. One important property of the DEB failure detection scheme is that collected data can be kept for later correlation purposes. This way, we can use time series of collected data to mitigate false positives and therefore minimize the use of costly error recovery schemes.

Another applicability of this stateful failure detector is similar to the approach by RAD Labs [2], where current run-time data is compared to previously collected data. This is used to trace critical paths through a computation and potentially detect anomalies and failures that would otherwise not be discovered by low-level failure detection schemes. For instance, a computer might respond to a ping indicating that everything is in order. However, higher-level application data can still not deliver its functionality within the expected time frame. For this purpose, multiple failure-detection modules have to interact and cooperate.

Extensive fine grained monitoring has two contradictory issues associated with it. First, the collected data can be used positively for important run-time management decisions. Second, monitoring might have negative impact on system performance as network bandwidth, storage, and processing capacity must be spent in order to collect, store, and process the monitored system metrics. To solve this scaling problem, we attempt to minimize network traffic by local monitoring and control if possible. Hence, monitored metrics can be kept local to a node if management decisions can be executed locally. For instance, a pipeline stage can monitor the size of its outbound queue. If the queue becomes full, the stage can slow down and choke back the predecessor in the pipeline, it can flag an alert to a remote management service, or more compelling, actually trigger creation of a replica of the pipeline bottleneck.

This decentralized combination of local monitoring combined with associated policy decisions is what we denote a proactive failure detector. Proactivity captures the notion that policy decisions can be taken prior to the appearance of a potential failure. This is again based on local suspicions reported by the local failure detector. The net effect of this in-network management structure is reduction of data on the wire and related latency problems.

We argue for monitoring of data through the whole vertical software layer stack. This includes application-level monitoring, or more accurately put, application-level failure detection. Closely related to this end-to-end argument inspired approach, is that applications also treat failures as a first order design consideration. So far, non-functional requirements of applications have as a rule often been supported by the underlying run-time. We do not intend to supplement this standard approach, but rather complement it with specific policy decisions embedded in the application itself. Again, this is based on the principle that some error recovery actions are better handled by the application itself.

Our autonomous run-time is surprisingly simple in complexity, especially when related to the powerful autonomic properties it provides. This simplicity stems from the computational model we deliberately chose, where a distributed computation is modelled as a directed acyclic graph of functional transformation stages. These are interconnected through outbound queues polled (and consumed) by one or more successor stages. One can consider each such functional stage as one step in a computation organized in consecutive steps that makes progress one after the other. This lends itself naturally to a checkpoint/resume scheme for error recovery, or to replication schemes solving failures at run-time. Similarly, this scheme also lends itself naturally to a dynamic incremental growth provisioning model. Adding extra capacity can now be done without human intervention in most situations.

5 Related Work

The inner workings of several large-scale enterprise systems illustrate well the complexity of enterprise systems [4, 5, 8]. Although the DEB run-time shares many similar functions with these systems, we incorporate the principle of autonomous computing to a greater extent.

Techniques for autonomous computing have previously been used within the realm of Peer-to-Peer (P2P) computing. For instance, P2P routing substrates self-organize to maintain routing invariants during changes in the overlay topology [16]. Storage networks self-heal by adding new object replicas when storage nodes fail [7]. Such techniques are also applicable in DEB. However, each P2P overlay network is designed to provision for only a small range of applications and can in isolation not efficiently support the wider range of functions found in an enterprise setting.

We structure our computations similarly to how SEDA computation are organized [17]. In SEDA, applications are structured as stages connected by explicit event (communication) queues. The SEDA architecture is typically demonstrated at a single site like, for instance, a web server, while our applications are intended to span many computers.

There exists an extensive body of theoretical work on failure detector oracles [3]. In practice, failure detectors within the fail-stop model use variants of pinging or heartbeats in combination with timeouts to distinguish correct processes from crashed ones. The Fireflies protocol [10] avoids the problems of a static global time-out by adjusting time-outs locally according to measured packet-loss rates. Algorithms for detecting certain Byzantine failures have also been proposed [6, 11]. Although the DEB platform can accommodate such failure detectors, we currently do not consider the full range of Byzantine faults. However, as argued by Doudou et al. [6], the notion of Byzantine faults is intimately associated with the overlying application, which we address in DEB through application specific SLAs.

A similar scheme to our application-level failure handling has previously been implemented in NAP, our fault-tolerance protocol for mobile agents [9]. In NAP, a mobile agent has left behind a rear-guard agent (or several) that acts as a failure detector. Upon detecting a failure, the rear-guard agent performs application specific recovery actions. The Dynamo storage system by Amazon [5] uses a similar recovery approach with application-assisted conflict resolution.

6 Conclusion

Our novel contributions is as follows. First, we propose a stateful, proactive failure detection scheme that is embedded in a run-time for information access applications. This is used to implement autonomic behavior like self-configuration, self-optimization, and self-healing. Self-healing is harder than one might consider at a first glance, because an autonomous fault-tolerant system should be able to repair itself independently. Having a too eager failure-detection mechanism then, might trigger costly error recovery actions too early. A stateful one can be used to correlate current events with historic events to detect anomalies more accurately.

Next, we demonstrate that our extensible, pipelined programming model can be implemented and deployed on a cluster of computers with autonomic properties supported. We demonstrate through experiments that, for instance, load surges can be dynamically handled by incrementally adding resources. This happens transparently while the computation is running.

Also, we propose application-level failure detection as a concept. Such a detector allows applications to identify more holistic application-level failures, be it that portions of the application are saturated or that end-to-end SLAs are violated. Finally, we propose failure recovery actions as a first order design consideration for high-availability systems.

Acknowledgements

The authors would like to thank Krister Mikalsen at Fast Search & Transfer for useful insights into requirements for future enterprise software systems. Also, thanks to Åge Kvalnes and the anonymous reviewers for helpful comments.

References

[1] E. M. Atkins, T. F. Abdelzaher, K. G. Shin, and E. H. Durfee. Planning and resource allocation for hard real-time, fault-tolerant plan execution. In *Proceedings of the 3rd Annual Conference on Autonomous Agents*, pages 244–251. ACM, 1999.

[2] P. Bodic, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, and D. Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Proceedings of the 2nd International Conference on Autonomic Computing*, pages 89–100. IEEE, June 2005.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 137–150. USENIX, Dec., 2004.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st Symposium on Operating Systems Principles*. ACM, Oct. 2007.

[6] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings of the 3rd European Dependable Computing Conference*, volume 1667 of *Lecture Notes on Computer Science*, pages 71–87. Springer-Verlag, 1999.

[7] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 75–80. IEEE, May 2001.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles*. ACM, 2003.

[9] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical fault-tolerance for itinerant computations. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 180–189. IEEE, 1999.

[10] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proceedings of the 1th EuroSys Conference*, pages 3–13. ACM, Oct. 2006.

[11] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, Jan. 2003.

[12] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia distributed monitoring system: Design, implementation and experience. *Parallel Computing*, 30:817–840, June 2004.

[13] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[14] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[15] J. F. Shoch and J. A. Hupp. The "worm" program—early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, Mar. 1982.

[16] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

[17] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 230–243. ACM, Oct. 2001.

[18] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart. An architectural approach to autonomic computing. In *Proceedings of the 1th International Conference on Autonomic Computing*, pages 2–9. IEEE, 2004.