Autonomic Computing–A System-Wide Perspective[*]

Robbert van Renesse Kenneth P. Birman

Department of Computer Science
Cornell University, Ithaca, NY 14853
{*rvr, ken*}*@cs.cornell.edu*

Autonomic computing promises computer systems that are self-configuring and self-managing, adapting to the environment in which they run, and to the way they are used. In this chapter, we will argue that such a system cannot be built simply by composing autonomic components, but that a system-wide monitoring and control infrastructure is required as well. This infrastructure needs to be autonomic itself. We demonstrate this using a datacenter architecture.

1 Introduction

Computer systems develop organically. A computer system usually starts as a simple clean system intended for a well-defined environment and applications. However, in order to deal with growth and new demands, storage, computing, and networking components are added, replaced, and removed from the system, while new applications are installed and existing applications are upgraded. Some changes to the system are intended to enhance its functionality, but result in loss of performance or other undesired secondary effects.. In order to improve performance or reliability, resources are added or replaced. The particulars of such development cannot be anticipated; it just happens the way it does.

Organic development seldom leads to simple or optimal systems. On the contrary, an organically developing system usually becomes increasingly complex and difficult to manage. Configurations of individual components become outdated. Originally intended for a different system, they are no longer optimal for the new environment. System documentation also becomes outdated, and it is difficult for new system managers or application developers to learn how and why a system works. While the original system was designed with clear guidelines, an organically developed system appears ad hoc to a newcomer.

The autonomic computing effort aims to make systems self-configuring and self-managing. However, for the most part the focus has been on how to make system components self-configuring and self-managing. Each such component has its own feedback loop for adaptation, and its own policies for how to react to changes in its environment. In an organic system, many of such components may be composed in unanticipated ways, and the composition may change over time in unanticipated ways. This may lead to a variety of problems, some of which we now address.

[*]The authors were supported by grants from the Autonomic Computing effort at Intel Corporation, from DARPA's Self Regenerative Systems program, and from NSF through its TRUST Science and Technology Center.

First, the components, lacking global control, may oscillate. Consider, for example, a machine that runs a file system and an application that uses the file system. Both the file system and the application are adaptive—they both can increase performance by using more memory, but release memory that is unused. Now apply a load to the application. In order to increase request throughput, the application tries to allocate more memory. This memory is taken from the file system, which as a result cannot keep up with the throughput. Due to decreased throughput, the application no longer requires as much memory and releases it. The file system can now increase its throughput. In theory, an autonomic allocation scheme exists so that these components may converge to an optimal allocation of memory between them. However, if the components release the same amount of memory each time, they are more likely to oscillate and provide sub-optimal throughput. While with two components and one shared resource such a situation is relatively easy to diagnose and correct, with more components and resources doing so becomes increasingly hard.

Second, even if all components of a system are autonomic, the configuration of the system as a whole is typically done manually. For example, some of the machines may be used to run a database, while other machines may be dedicated to run web servers or business logic. Such partitioning and specialization reduces complexity, but likely results in non-optimal division of resources, particularly as a system changes over time. Perceived inefficiencies are usually solved by adding more hardware for specific tasks; for example, more database machines may be added. In a system spanning multiple time zones, performance critical data or software licenses may need be moved over time. Such changes often require that the configuration of many other machines have to be updated, and finding and tracking such dependencies is non-trivial.

Perhaps the most difficult problem is the one of scale. As an organic system grows in the number of resources, applications, and users, its behavior becomes increasingly complex, while individual components may become overloaded and turn into bottlenecks for performance. Few if any system administrators will have a complete overview of the system and its components, or how its components interact. This not only makes it more difficult to manage a system, but it also becomes harder to develop reliable applications. Installing new resources or applications may break the system in unexpected ways. To make matters even more complicated, such breakage may not be immediately apparent. By the time the malfunction is observed, it may no longer be clear which system change caused the problem.

In this chapter, we argue that a system-wide autonomic control mechanism is required. We describe a Scalable Monitoring and Control Infrastructure (SMCI) that acts as a system-wide feedback loop. The infrastructure allows administrators to view their system and zoom into specific regions with directed queries, and also allows administrators to change the behavior of the system. Administrators can create global policies for how components should adapt. While it is still necessary for the individual components to adapt their behavior, the infrastructure guides such local adaptation in order to control system-wide adaptation. This way, internal oscillation may be all but eliminated, and the infrastructure can control how applications are assigned to hardware in order to maximize a global performance metric.

2 Scalable Monitoring and Control Infrastructure

An SMCI can be thought of as a database, describing the underlying managed system. For convenience, the database reflects an organizational hierarchy of the hardware in the system. We call the nodes in such a hierarchy domains. For example, a building may form a domain. Within a building, each machine room forms a domain. Each rack in the machine room forms a subdomain. Associated with each domain is a table with a record for each subdomain. The records reflect and control the subdomains. See Figure 1 for an example.

For example, in the table of the domain of a rack, there may be a record for each machine, specifying attributes like the CPU type, the amount of memory, its peripherals, what applications it is running, and the load on the machine. Some attributes are read-only values, and can only be updated by the machine itself. An example is the number of applications that is running on the machine. Other attributes may be written by external sources in order to change the behavior of the component. For example, the machine record may contain attributes that govern relative priorities of the applications that it is running.

The tables look like database relations, and would typically be accessed using a query language like SQL. An SMCI will allow one-shot queries like: "how many machines are there?" as well as continuous standing queries like "inform me of updates to the number of machines." The latter query works in conjunction with a publish/subscribe or event notification mechanism, and publishes updates to interested parties.

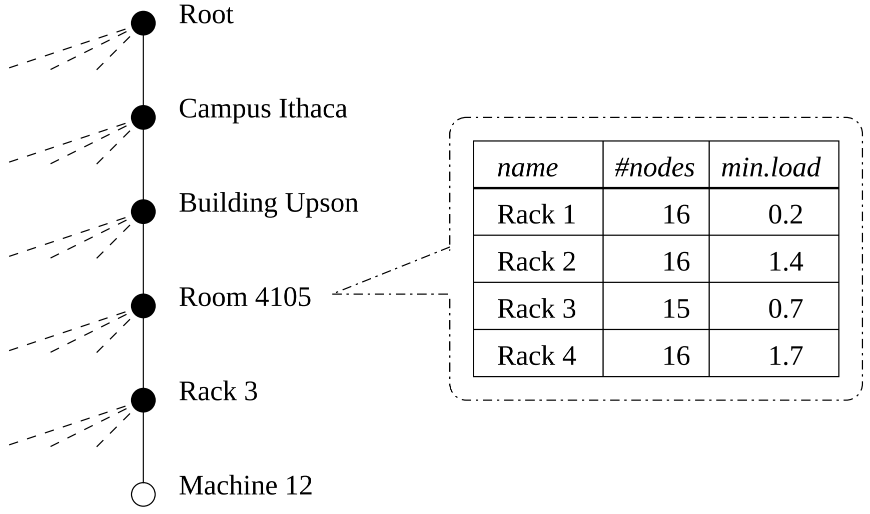
A unique feature of SMCIs as compared to ordinary relational databases is their ability to do "hierarchical aggregation queries." An ordinary aggregation query is limited to a single table, while a hierarchical query operates on an entire domain, comprising a tree of tables. For example, one may ask, "How many machines are there in room 4105?" This query adds up all the machine in all the racks in room 4105.

Note that unlike a database an SMCI is not a true storage system. It gathers attributes from system components and can update those attributes, but it does not store them anywhere persistently. It may store them temporarily for caching purposes. In other words, the tables observed in an SMCI are virtual, and dynamically materialized as necessary in order to answer queries about the system. In order to do so, an SMCI does need to keep track of the organizational hierarchy of the system as well as the standing queries. In addition, an SMCI has to detect and reflect intentional reconfigurations as well as changes caused machine crashes or network link failures.

An SMCI allows clients, which may include both administrators and running programs, to ask various questions about the system. Examples of such questions include "where is file foo," or, "where is the closest copy of file foo?" "Which machines have a virus database with version less than 2.3.4?" "Which is the heaviest loaded cluster?" "Which machine in this cluster runs a DNS resolver?" "How many web servers are currently up and running?" "How many HTTP queries per second does the system receive as a whole?" "Which machines are subscribed to the topic bar?" And so on.

The real power of the system derives from the fact that the query results may be fed back into the system in order to control it. This creates an autonomic feedback loop for the system as a whole. It can drive the configuration of the system, and have the system adapt automatically to variations in its environment and use, as well as recover from failures. For example, the number of replicas of a service may depend on the measured load on the service. The replica placement may depend on where the demand originates and the load on the machines. All this may be expressed in the SMCI.

In order for an SMCI to work well, it needs to be autonomic itself. Should an SMCI require significant babysitting compared to the system it is managing, it would cease to be useful. An SMCI should



name	*#nodes*	*min.load*
Rack 1	16	0.2
Rack 2	16	1.4
Rack 3	15	0.7
Rack 4	16	1.7

Figure 1: An example of a SMCI hierarchy. On the right is the domain table of the room 4015, which has four racks of machines. Each rack has three attributes: its name, the number of operational machines in the rack, and the minimum load across the machines.

be robust and handle failures gracefully. It should be self-configuring to a large extent, and provide accurate information using a minimum of network traffic. It should scale well, allowing for the hierarchy to be grown as new clusters, machine rooms, or buildings are added to the organizational structure.

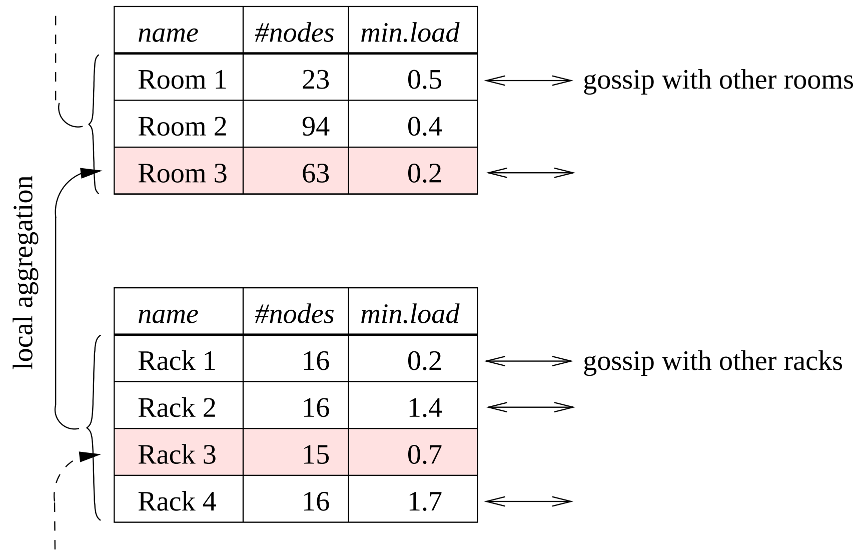
Our choice of language deliberately suggests that a system would often have just one SMCI hierarchy, and indeed for most uses, we believe one to be enough. However, nothing prevents the user from instantiating multiple, side-by-side but decoupled, SMCI systems, if doing so best matches the characteristics of a given environment.

A number of SMCIs have been developed, and below we give an overview of various such systems. All are based on hierarchies. Most use a single aggregation tree. Examples are Captain Cook [9], Astrolabe [10], Willow [11], DASIS [1], SOMO [13], TAG [5], and Ganglia [8, 6]. SDIMS [12] and Cone [2] use a tree per attribute. Below we give an example of two peer-to-peer SMCIs, one that use a single tree, and one that uses a tree per attribute.

2.1 Case Study: Astrolabe

The Astrolabe system [10] is a peer-to-peer SMCI, in that the functionality is implemented in the machines themselves and there are no centralized servers to implement the SMCI. Doing so improves the autonomic characteristics of the SMCI service. In Astrolabe, each machine maintains a copy of the table for each domain it is a member of. Thus it has a table for the root domain and every intermediate domain down to the leaf domain describing the machine itself. Assuming a reasonably balanced organizational, the amount of information that a machine stores is logarithmic in the number of machines.

The attributes of a leaf domain in Astrolabe are attributes of the machine corresponding to the leaf



name	*#nodes*	*min.load*
Room 1	23	0.5
Room 2	94	0.4
Room 3	63	0.2

name	*#nodes*	*min.load*
Rack 1	16	0.2
Rack 2	16	1.4
Rack 3	15	0.7
Rack 4	16	1.7

Figure 2: Data structure of a machine in Rack 3 inside Room 3 of some building. The machine generates the attributes of Rack 3 and Room 3 using local aggregation. The attributes of peer racks and rooms are learned through gossip.

domain, and may be writable. Standing aggregation queries generate the attributes of non-leaf domains (so-called internal domains), and are strictly read-only. The table of a domain is replicated on all the machines contained in that domain, and kept consistent using a gossip protocol.

Using an organizational hierarchy based on DNS as an example, consider the leaf domain fox.cs.cornell.edu, and consider how this machine may generate the attributes of the cornell.edu domain has subdomains cs, bio, chem, etc. The machine fox.cs.cornell.edu can generate the attributes of the cs.cornell.edu domain locally by aggregating the attributes in the rows of its copy of the domain table of cs.cornell.edu. In order to learn about the attributes of bio.cornell.edu and chem.cornell.edu, the machine has to communicate with machines in those domains.

In order to do so efficiently, Astrolabe utilizes its mechanisms to manage itself. Astrolabe has a standing query associated with each domain that selects a small number, say 3, of machines from that domain. The selection criteria do not really matter for this discussion, and can be changed by the system administrator. These selected machines are called the representatives of the domain. Peer representatives gossip with one another, meaning that they periodically select a random partner and exchange their versions of the domain table. The tables are merged pair-wise (Astrolabe uses a timestamp for each record in order to determine which records are most recent). Applying this procedure with random partners results in an efficient and highly robust dissemination of updates between representatives. The representatives gossip the information on within their own domains.

Thus, fox.cs.cornell.edu computes the attributes of the cs.cornell.edu domain locally, and learns the attributes of bio.cornell.edu etc. through gossip. Applying this strategy recursively, it can then compute the attributes of cornell.edu locally and learn the attributes of the peer domains of cornell.edu through gossip. Using this information it can compute the attributes of the edu domain. Note, however, that there is no consistency guaranteed. At any point in time, fox.cs.cornell.edu and lion.cs.cornell.edu may have different values for the attributes in their common domains, but the gossip protocols disseminate updates quickly leading to fast convergence.

2.2 Case Study: SDIMS

Another peer-to-peer approach to an SMCI is taken by SDIMS (Scalable Distributed Management Information System) [12]. SDIMS is based on Plaxton's scheme for finding nearby copies of objects [7] and also upon the Pastry Distributed Hash Table, although any Distributed Hash Table (DHT) that employs bit-correcting routing may be used.

In a DHT, there is a key space. Each machine has a key in this key space, and maintains a routing table in order to route messages, addressed to keys, to the machines with the nearest-by keys. For each position in a key, a machine maintains the address of another machine. For example, supposed that keys consist of 8 bits and a particular machine has key 01001001. If this machine receives a message for key 01011100, the machine determines the common prefix, 010, and finds in its routing table the address of a machine with address 0101xxxx. It then forwards this message to that machine. This continues until the entire key has been matched or no machine is found in the routing table.

In SDIMS, the name of an attribute is hashed onto a key. The key induces a routing tree. This tree is then used to aggregate attributes of the same name on different machines. Unlike Astrolabe, SDIMS has a flexible separation of policy and mechanism that governs when aggregation happens. In SDIMS, one can request that aggregation happens on each read, in which case the query is routed from the requester to the root of the tree, down to the leaves in the tree, retrieving the values, back up the tree aggregating the values, and then down to the requester in order to report the result. For attributes that are read rarely, this is efficient. For attributes that are written rarely, SDIMS also provides a strategy that is triggered on each write, forwarding the update up the tree and down to subscribers to updates of the aggregate query. SDIMS provides various strategies in between these extremes, as well as a system that automatically tries to determine which is the best strategy based on access statistics.

In order to make Pastry suitable for use as an SMCI, several modifications were necessary. Most importantly, the original Pastry system does not support an organizational hierarchy, and cannot recover from partition failures.

2.3 Discussion

Because SDIMS utilizes a different tree for each attribute, and optimizes aggregation for each attribute individually, it can support more attributes than Astrolabe. On the other hand, Astrolabe can support multi-attribute queries in a straightforward way. Astrolabe's gossip protocols are highly robust and work even in partially connected systems, but are less efficient than SDIMS's protocols.

Both Astrolabe and SDIMS are self-configuring, self-managing protocols. Each requires that each node is given a unique name, and an initial set of "contact nodes" in order to bootstrap the system. After that, failure detection and recovery, and membership management in general is taken care of automatically.

3 Service-Oriented Architecture

Many large datacenters, including Google, eBay, Amazon, Citibank, etc., as well as at datacenters used in government, organize their systems as a Service-Oriented Architecture (SOA). In [4], the authors introduce a common terminology for describing such systems. Below we will review this terminology, and give an example of how a web-based retailer might create a self-configuring and self-managing datacenter using a SMCI.

A collection of servers, applications, and data at a site is called a *farm*. A collection of farms is called a *geoplex*.

A service may be either *cloned* or *partitioned*. Cloning means that the data is replicated onto a collection of nodes. Each node may provide its own storage (inefficient if there are many nodes), or use a shared disk or disk array. The collection of clones is called a *Reliable Array of Cloned Services* or *RACS*. Partitioning means that the data is partitioned among a collection of nodes. Partitions may be replicated onto a few nodes, which then form a *pack*. The set of nodes that provide a packed-partitioned service is called a *Reliable Array of Partitioned Services* or *RAPS*.

While a RAPS is functionally superior to a RACS, a RACS is easier to build and maintain, and therefore many datacenters try to maximize the use of the RACS design. RACS are good for read-only, stateless services such as often found in the front-end of a datacenter, while RAPS are better for update-heavy state as found in the storage back-end. A system built as a RAPS in which each services is a RACS potentially combines the best of both options, but can be complex to implement.

SOA principles are being applied in most if not all large datacenters. All these systems have developed as described in the introduction, in an organic way, starting with a few machines in a single room. Now they are geoplexes with thousands to hundreds of thousands machines, with hundreds or thousands of applications deployed. Literally billions of users depend on the services that these systems provide.

The configuration and management of such systems is a daunting task. Problems that result in blackouts or brown-outs need to be resolved within seconds. Automation is consequently highly desired.

The state-of-the-art is that each service has its own management console. Yet the deployed services depend on one another in subtle or not so subtle ways. If one service is misbehaving, this may be caused by the failure of another service. Obtaining a global view of what is going on and quickly locating the source of a problem is not possible with the monitoring tools currently provided. Also, while such services may have some support for self-configuration and self-management built-in, doing so on a global scale is not supported.

A SMCI can be used to solve such problems. A SMCI has a global presence, and can monitor arbitrary sensory data available in the system. Subsequently, the SMCI can provide a global view by installing an appropriate aggregation query. While useful to a system administrator, such data can also be fed back into the system help automate the control of resource allocation or drive actuators, rendering the system self-configuring and self-managing. We will demonstrate this for a web-based retailer.

4 An Example

A web-based retailer's datacenter typically consists of three tiers. The front-end consists of web servers, and handles HTTP requests from remote clients. The web servers issue parallel calls to services implemented by applications running in the middle tier. A web page typically requires calls into various services for the various sections of the page. Examples

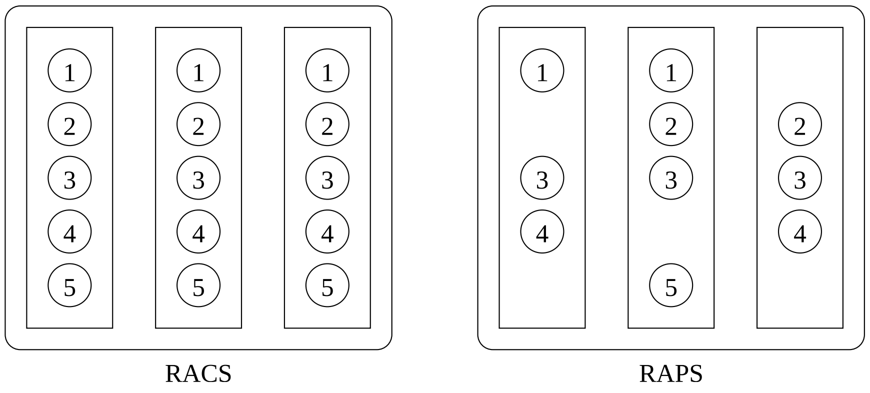


Figure 3: Example of RACS and RAPS architectures. In RACS, each object is replicated on all three machines. In RAPS, an object is replicated only on a subset of the machines, allowing more objects to be stored and offering greater flexibility in availability guarantees.

of services include an HTTP Session Service, product search and browsing, various services that describe a product (pictures, technical data, availability, pricing, etc.), a service that recommends related products, and many more. The services may need to call on one another. Long-term storage is handled by the back-end tier.

The front-end servers typically are stateless. For every incoming request they simply call into the second tier. A RACS architecture is an obvious choice: each web server is essentially identical, and maintains little data. The third tier's storage reflects the actual assets held by the retailer, and also deals with credit card information and transactions, and therefore needs to be highly reliable and consistent. A RACS architecture would not scale, as updates would have to be applied, consistently, at each server. Consequently, the third tier is typically implemented as a RAPS. For the middle tier, a choice between RACS and RAPS can be made on a per-service basis.

The description so far is a logical depiction of a datacenter. Physically, the datacenter consists of a collection of machines, disks, networks, and often, load balancers that distribute requests among nodes of a service. Both the logical and physical aspects of a datacenter are in flux. Physically, machines die, are added, or are replaced, while the network architecture may also change frequently. Logically, many datacenters deploy several new applications per week, and retire others. As a result, the configuration of a datacenter must be updated continuously, and more frequently as a datacenter grows.

The web servers in the front-end need to know the layout of a web page, which services it should invoke for the various sections, and how it should contact these services so as the balance the load across the nodes that comprise the service, while avoiding nodes that are faulty or overloaded. The services in the middle tier depend on other services in the middle and third tier, and also need to know how to contact those services. The number of machines assigned to a service depend on the load distribution among the machines. Which machines are assigned to which applications depends upon data placement, and it may be necessary to move data when the sys-

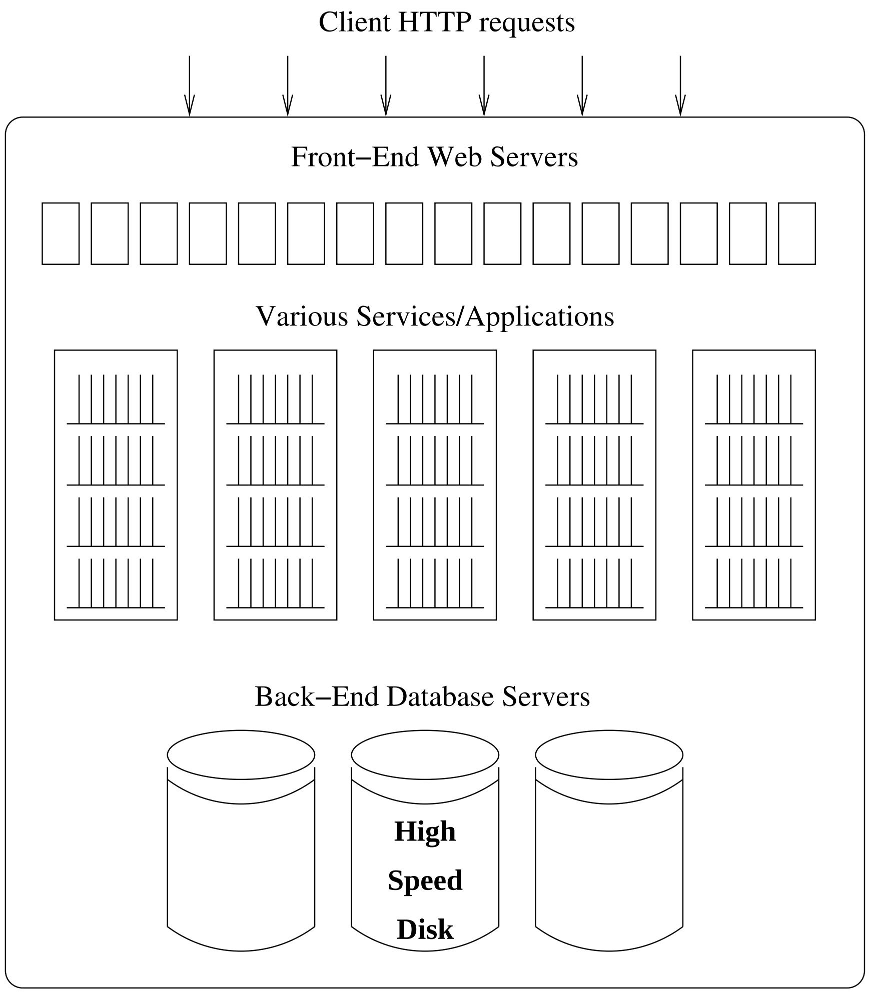


Figure 4: A datacenter is typically organized as three tiers.

tem is reconfigured. We will now show how a SMCI can be used in such an environment.

A simple SMCI organization for this system could be as follows. The root of the tree would describe a geoplex, with a child node for each farm. Each farm would have a child node for each cluster of machines. For each cluster, there would be a child node for each machine in the cluster. Each machine would be able to report what applications it is running, as well as various information about these applications.

4.1 Using a RACS

As an example, consider the Session Service which keeps track of sessions with web clients. When an HTTP request arrives at a web server, the web server contacts the Session Service in order to obtain information about the current session. In order to do so, the web server has to find a machine that runs the Session Service (called a Session Server henceforth) and knows about the session. If the Session Service is organized as a RACS, any machine will do. If it is organized as a RAPS, only some of the machines in the RAPS know about the session. For now we assume the Session Service is organized as a RACS.

The system has permanently installed an aggregation query in the SMCI that counts the number of Session Servers in a SMCI domain (1 for the leaf domains that represent the machines that run the Session Service). Starting in the root domain, the web server would first locate the child node corresponding to its own farm. Then it would locate a child node (a cluster) that has more than Session Server. If there is more than one such cluster, it could pick one at random, weighted by the number of Session Servers for fairness. Finally, it would pick one of the Session Servers at random from the cluster. Additionally, the Session Servers could report their load. Using an aggregation function that reports the minimum load in a domain, web servers could locate the least loaded Session Server.

4.2 Using a RAPS

Should the Session Service be implemented as a RAPS, then there are various options in order to locate one of the Session Servers in the pack for the given session identifier. One possibility is that the Session Servers themselves maintain such a map. A request from a web server is first sent to an arbitrary Session Server, and then forwarded to the correct one. This delay can be avoided by using a Bloom filter [3]. A Bloom filter is a concise representation of a set in the form of a bitmap.

Each Session Server would report in a Bloom filter the set of sessions they are responsible for. An aggregation function would simply 'bitwise or' these filters together in order to create a Bloom filter for each domain. As a result, each domain would report what sessions it is responsible for. Using this information, a web server can quickly find a Session Server in the pack responsible for a particular session.

So far we have described how the web servers can use the SMCI in order to find Session Servers, and this is part of how the datacenter manages itself. As another example of self-management, the Session Service can use the SMCI to manage itself. We will focus on how Session Servers in a RAPS architecture choose which sessions they are responsible for. This will depend on load distribution. Complicating matters, machines may be added or removed. We also try to maintain an invariant of k replicas for each session, no less, but also no more.

Each Session Server can keep track of all the machines in the Session Service simply by walking the SMCI domain tree. Using the Session Service membership, it is possible to apply a deterministic function that determines which machines are responsible for which sessions. For example, a session could be managed by the k machines with the lowest values of $\mathcal{H}(\textit{machine ID}, \textit{session ID})$, where $\mathcal{H}$ is a hash function. These machines would be responsible for running a replication protocol to keep the replicas consistent with one another.

5 Conclusion

A large autonomic computing system cannot be composed from autonomic components. The configuration of the components would not self-adapt, the composition could oscillate, and individual components might become bottlenecks. In order to have system-wide autonomy, a system-wide feedback loop is necessary. We described the concept of a Scalable Monitoring and Control Infrastructure, and how it may be applied in an autonomic Service Oriented Architecture.

References

[1] K. Albrecht, R. Arnold, and R. Wattenhofer. Join and leave in peer-to-peer systems: The DASIS approach. Technical Report 427, Dept. of Computer Science, ETH Zurich, November 2003.

[2] R. Bhagwan, G. Varghese, and G.M. Voelker. Cone: Augmenting DHTs to support distributed resource discovery. Technical Report CS2003-0755, UC, San Diego, July 2003.

[3] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *CACM*, 13(7):422–426, July 1970.

[4] B. Devlin, J. Gray, B. Laing, and G. Spix. Scalability terminology: Farms, clones, partitions, packs, racs and raps. Technical Report MSR-TR-99-85, Microsoft Research, 1999.

[5] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, Boston, MA, December 2002. USENIX.

[6] M.L. Massie, B.N. Chun, and D.E. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7), July 2004.

[7] C.G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[8] F.D. Sacerdoti, M.J. Katz, M.L. Massie, and D.E. Culler. Wide area cluster monitoring with Ganglia. In *Proc. of the IEEE Cluster 2003 Conference*, Hong Kong, 2003.

[9] R. van Renesse. Scalable and secure resource location. In *Proc. of the 33rd Annual Hawaii Int. Conf. on System Sciences*, Los Alamitos, CA, January 2000. IEEE, IEEE Computer Society Press.

[10] R. van Renesse, K.P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(3), May 2003.

[11] R. van Renesse and A. Bozdog. Willow: DHT, aggregation, and publish/subscribe in one protocol. In *Proc. of the 3rd Int. Workshop on Peer-To-Peer Systems*, San Diego, CA, February 2004.

[12] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *Proc. of the '04 Symp. on Communications Architectures & Protocols*, Portland, OR, August 2004. ACM SIGCOMM.

[13] Z. Zhang, S.-M. Shi, and J. Zhu. SOMO: Self-Organized Metadata Overlay for resource management in P2P DHT. In *Proc. of the Second Int. Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, February 2003.