DRACO: Distributed Resource-aware Admission Control for Large-Scale, Multi-Tier Systems

Domenico Cotroneo, Roberto Natella, Stefano Rosiello

DIETI - Università degli Studi di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy

Abstract

Modern distributed systems are designed to manage overload conditions, by throttling the traffic in excess that cannot be served through *overload control* techniques. However, the adoption of large-scale NoSQL datastores make systems vulnerable to *unbalanced overloads*, where specific datastore nodes are overloaded because of hot-spot resources and hogs.

In this paper, we propose DRACO, a novel overload control solution that is aware of data dependencies between the application and the datastore tiers. DRACO performs selective admission control of application requests, by only dropping the ones that map to resources on overloaded datastore nodes, while achieving high resource utilization on non-overloaded datastore nodes. We evaluate DRACO on two case studies with high availability and performance requirements, a virtualized IP Multimedia Subsystem and a distributed fileserver. Results show that the solution can achieve high performance and resource utilization even under extreme overload conditions, up to 100x the engineered capacity.

Keywords: Overload control; Traffic throttling; Hot-spot resources; Cluster systems; Network Function Virtualization

1. Introduction

Cloud computing is increasingly being adopted for running high-availability services in critical domains, such as telecom (e.g., the emerging *Network Function Virtualization* paradigm), healthcare, transportation, and more [1, 2, 3]. Capacity management is a key problem for these services, since they face high and variable volumes of traffic, and are challenged by *overload conditions*, e.g., in the case of major sports events, the launch of a new popular feature, and other mass events [4]. An overloaded system attempts to serve more traffic than its resources would allow, causing high resource contention and the violation of availability and latency goals. Moreover, applications under overload become prone to software failures due to race conditions, resource exhaustion, memory management bugs, and timeouts. Cloud elasticity alone does not suffice to achieve high availability (i.e., *five-nines* or more) required by critical domains [4], since scaling-out can take up to several minutes to allocate new VMs [5, 6], and can require coordination across several datacenters [7, 8]. Therefore, *overload control* is a common best practice to prevent failures, by limiting (*throttling*) the incoming user requests admitted to the system [4, 9, 10]. Ideally, overload control admits only a subset of user requests to utilize all of the system's engineered capacity, and drops the exceeding requests to prevent resource contention.

However, overload control is still a challenging problem in *multi-tier systems with large-scale datastores.* Modern services reach a massive scale, by organizing the *application tier* into hundreds of replicas deployed over multiple datacenter regions, e.g., using the Amazon AWS infrastructure as in the Netflix popular streaming platform [11]; and by keeping the state of the application, such as session information, multimedia resources, and other data in a separate *datastore tier.* This approach is enabled by the emerging *NoSQL datastores*, such as Memcached [12], which can balance the load across

Email addresses: `cotroneo@unina.it` (Domenico Cotroneo), `roberto.natella@unina.it` (Roberto Natella), `stefano.rosiello@unina.it` (Stefano Rosiello)

many nodes, through consistent hashing [13, 14]. For example, these solutions allow Netflix to scale production systems through tens of thousands of datastore instances, serving about 30 million requests per second [15, 16, 17].

The downside of this architecture is that it introduces *data dependencies* between the application and the datastore tiers. Data requests cannot be uniformly balanced across the datastore tier, but must be served by the specific nodes that hold the requested data. These dependencies are problematic for overload control purposes, since they make the system vulnerable to *unbalanced overloads*, i.e., overloads that affect specific nodes in the datastore tier [18, 19, 20]. A typical cause of unbalanced overloads are *hot-spots* [21, 22, 23, 24], i.e., multimedia resources that suddenly become popular. Since data requests have to be directed to those specific storage nodes that manage the hot-spot resource (e.g., based on a consistent hashing scheme), the overloaded caching nodes become a bottleneck for the entire system. Other causes of unbalanced overloads are over-commitment (e.g., the same host is shared by multiple tenants, which face a high load at the same time), and design and configuration bugs [18, 19, 20]. As discussed in the next section, existing throttling solutions are unaware of data dependencies, and cannot efficiently deal with unbalanced overload conditions.

In this paper, we propose *DRACO* (*Distributed Resource-aware Admission COntrol*), a novel data-dependency aware overload control solution, which can efficiently address unbalanced overload conditions that arise in the datastore tier. *DRACO* performs selective admission control, by tuning the amount and type of admitted traffic according to data dependencies among the tiers, and to the current capacity of individual nodes. The solution is designed to be deployed in front of the application tier, in order to avoid side effects on the application, and it is tailored for large-scale NoSQL datastores, in order to fit the architecture of modern multi-tier systems.

We experimentally evaluate *DRACO* on two case studies: a virtualized *IP Multimedia Subsystem*, which requires carrier-grade levels of performance and availability [25], and a *Distributed Fileserver*, which is very sensitive to overloads caused by hot-spot resources. We present an experimental evaluation on very high and unbalanced overload conditions, by generating service requests up to 100 times the maximum capacity that the system can manage. With our solution, the system is able to use more than 90% of its engineered capacity, without any latency violations for the admitted users, and no software failures due to resource exhaustion. Moreover, the solution is able to only drop the traffic in excess that uses the overloaded datastore nodes, thus enabling the full utilization of other datastore nodes.

In the following, Sec. 2 introduces overload control issues in multi-tier systems, and defines requirements for our solution. Sec. 3 presents the design of *DRACO*. Sec. 4 and 5 present the two case studies and experimental results. Sec. 6 discusses the overhead and scalability of *DRACO*. Sec. 7 discusses related work. Sec. 8 concludes the paper.

2. Unbalanced overloads in datastores

Large scale, multi-tier systems adopt load balancing schemes both on applications and on datastore tiers (Figure 1). On the application side, *service requests* are typically load-balanced by selecting an application node (a *replica*) from a pool of IP addresses, using DNS in a round-robin fashion. Then, application nodes generate one or more *data requests* towards the datastore tier. Service requests can be served by any application node without restrictions, since such nodes are stateless and independent from each other. Data requests, instead, cannot be processed by an arbitrary datastore node, since data resources are partitioned across datastore nodes, and the data required by the user is located only on one datastore node, or on a small subset (in the case of data replication). The location of data resources is typically identified using *consistent hashing* [13], which computes a lightweight, deterministic function that maps a *resource key* in the service request to datastore nodes. The dependencies to datastore nodes are based on the content of service requests from the clients and, thus, they are not known a-priori.

An overload condition occurs when a system has insufficient resources to serve the incoming requests, due to a bottleneck in one of its components. In particular, *unbalanced overload conditions* in the datastore nodes are quite challenging. One typical problem is represented by *hot-spots*: when users access specific resources much more frequently than others (for example, multimedia content or application that suddenly becomes popular on the web), the load on a subset of nodes will be higher. When

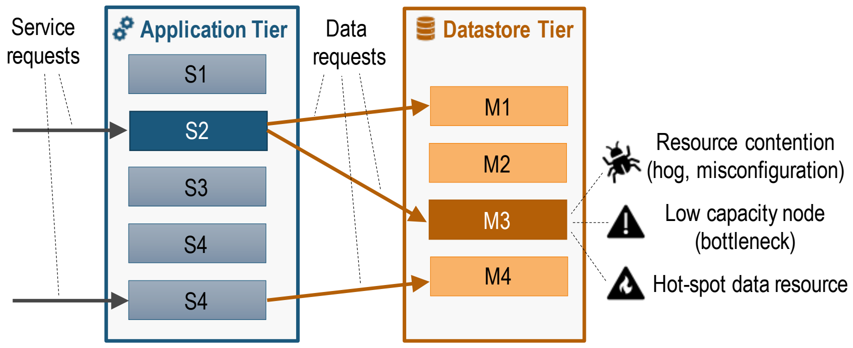


Figure 1: Overview of multi-tier systems.

datastore nodes are overloaded, the application services also become prone to failures. A symptom of a hot-spot is a highly skewed distribution of data requests to the nodes. For example, in Figure 1, the datastore node *M3* holds a hot-spot resource while the load on other nodes is within their capacity; in turn, the service request at application node *S2* (and any other service request that needs *M3*'s data) will experience a slow-down. Data consistency is also put at risk: if *S2* is writing data on both *M1* and *M3*, only the data in *M1* may be actually stored.

It should be noted that even in the ideal case where the datastore load is uniformly balanced, unbalanced overloads can still affect the datastore nodes. Indeed, they may have different configurations; they can be deployed on machines with different characteristics; or, the physical machines can be shared among different tenants. As a consequence, datastore nodes can exhibit different capacity. This means that unbalanced overloads can affect specific nodes in the datastore tier.

The definitive solution to unbalanced overloads is to redistribute and replicate data across datastore nodes, and to scale-out the nodes. However, since these solutions still leave the system vulnerable to overloads in the short-term, system designers also adopt overload control solutions. Despite that virtual machines and containers can be automatically and quickly provisioned, scale-out actions can still take up to several minutes, due to initialization and warm-up of application software [5, 6]. For example, this is an issue for stateful components, such as legacy, monolithic software still running in telecom and other critical infrastructures [26, 27], and for datastore technology, where the new replica first needs to load the data in-memory [28, 18]. While scale-out is taking place, the system is still prone to service failures (e.g., timed-out requests), which is an issue for high-availability services such as NFV [29].

To mitigate the unbalanced overloads in the short term, the system needs to throttle the volume of data requests towards overloaded datastore nodes. NoSQL datastores allow system designers to throttle the incoming data requests *between the application and the datastore tiers* [30, 31]. Unfortunately, throttling data requests is often not acceptable for developers, since the application tier would experience side effects, such as exceptions and data unavailability errors, which would increase the risk of application failures. For example, this can violate data consistency (e.g., by breaking transactions that span over several datastore nodes), or it can trigger complex and wasteful roll-backs of transactions.

Therefore, the only viable option is to throttle service requests at the application tier, *before* they enter into the system [32, 25, 33, 34, 35]. However, existing throttling solutions do not take into account data dependencies, and are still not suitable for addressing unbalanced overload conditions. They cause a load decrease across all nodes in the datastore tier, including the ones (a majority) that still have resources available for serving requests, leading to unnecessary waste of resources and service unavailability. For example, in Figure 1, if service requests from *S2* and *S4* are both rejected, the system's resources would be under-utilized, since *M4* would not be serving requests from *S4* despite it has still available capacity.

In summary, overload control needs to meet the following requirements to efficiently mitigate unbalanced overloads:

1. **Account for the actual capacity of nodes**

at run-time. Overload control should dynamically take into account the current (unbalanced) available capacity of datastore nodes, to handle hot-spots and transient capacity variations caused by hogs.

2. **Integrity of application service requests**. Overload control should throttle service requests in excess before they enter the application tier (i.e., not in the middle of service request processing); throttling in the datastore tier (i.e., after service request processing already started) would cause data consistency issues and waste of resources.

3. **High utilization of non-overloaded nodes**. Overload control should selectively reject service requests that use overloaded datastore nodes, while admitting service requests that only use non-overloaded nodes.

3. The proposed solution

The main novel feature of *DRACO* is the ability to mitigate unbalanced overload conditions that arise from a subset of nodes in the datastore tier, and at the same time to achieve high utilization of the non-overloaded parts of the datastore tier. The driving idea is to take advantage of knowledge of the application logic, in order to map service requests to the datastore nodes that are needed to serve the requests; and to only admit into the system a selected subset of them, by rejecting the ones that would attempt to access overloaded datastore nodes.

The solution adopts a distributed architecture, in order to scale with the size of the tiers; and it is designed to filter traffic at the application tier, that is, it avoids filtering the traffic at the datastore tier, which would cause inconsistencies between the application and storage tiers, and among the nodes of the storage tier. In the following, we introducing the architecture and components of the solution (Figure 2). In Sec. 4 and 5, we will implement and evaluate the solution in the context of two case studies. Section 6 will discuss the overhead and scalability of the proposed solution.

The first component of the solution is the *Distributed Memory* block. This component keeps track of the *location of resources* across datastore nodes, and of the *available capacity* of the nodes. The available capacity is an estimate of *the number of data requests that a datastore node can serve*, based on the recent history of data requests previously served by datastore nodes, and the amount of physical resources consumed when these requests were served. This information is periodically collected by a *Capacity Monitoring* component deployed in the datastore tier.

The residual capacity gives an indication of how many data requests are in excess in the storage tier, and it is used to identify which service requests should be rejected at the application tier. The actual throttling is performed by the *Distributed Admission Control* component, which acts as a tunnel between clients and application nodes. The admission decision is done by inspecting the service request, and by checking if there is enough residual capacity in both the current application node and in all of the storage nodes that are needed to process the service request.

3.1. Distributed Memory

The *Distributed Memory* component includes a datastore that handles the following two types of data (Figure 2):

- **Node Capacity Status**: For every datastore node, our solution has a counter representing the available capacity, in terms of *number of data requests* that the node can sustain in the current time window. This information is updated by the *Capacity Monitoring* block at the beginning of a new time window. The *Admission Control* block checks whether a service request can be served, given the available capacity of datastore nodes. Then, it decrements the available capacity upon acceptance of a service request.

- **Data Location Cache**: It is the location of data resources accessed by service requests. It is added, retrieved, and updated during the *Data Location Discovery* phase by the *Admission Control* block.

The *Distributed Memory* stores the Node Capacity Status in key-value pairs (e.g., indexed by the hostname or IP address of datastore nodes). The *Data Location Cache* stores a key-value pair for each service request. The key, which is application-dependent, represents a specific service request. The value is an array of integers, with one element for each datastore node, which represents the number of data requests to perform on each datastore node for the service request.

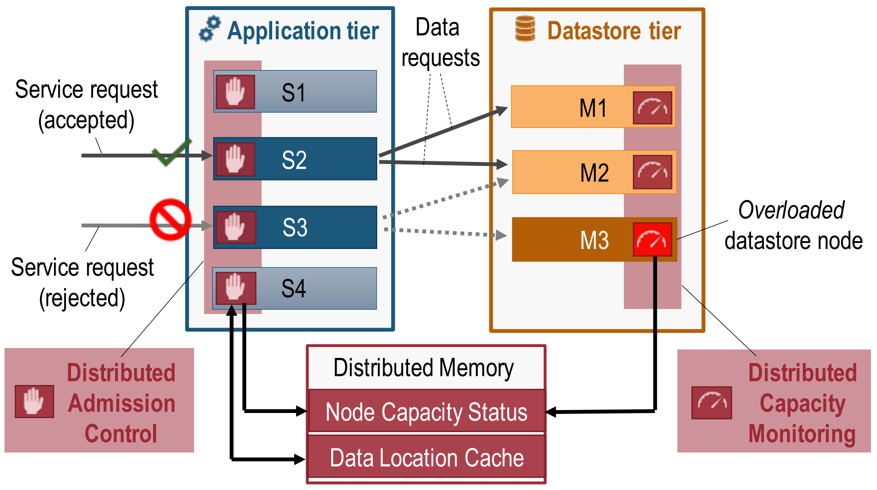

Application tier
S1
S2
S3
S4
Data requests
Datastore tier
M1
M2
M3
Service request (accepted)
Service request (rejected)
Overloaded datastore node
Distributed Admission Control
Distributed Memory
Node Capacity Status
Data Location Cache
Distributed Capacity Monitoring

Figure 2: Overview of *DRACO* architecture.

We leverage a distributed NoSQL datastore (such as Memcached) for the *Distributed Memory*. This technology allows the solution to handle data from a large number of nodes in the cluster, and simplifies the collection and the distribution of monitoring data. The *Distributed Memory* can be deployed either on the existing tiers, or in a dedicated tier. To avoid performance bottlenecks, the *Distributed Memory* establishes a fixed pool of persistent connections with the other nodes in the system. Thus, the number of connections only grows linearly with the number of nodes in the cluster (i.e., we do not introduce any direct communication between application and datastore nodes). Moreover, the capacity of the *Distributed Memory* can be scaled-up by distributing the monitoring data across several nodes. In Section 6, we discuss in detail the computational and communication costs of the *Distributed Memory*, and its scalability.

The *Distributed Memory* block stores the location of data resources accessed by past service requests. This caching improves the performance of the overload control solution, especially in the case of hot-spot resources, since information on hot-spot resources (which are repeatedly accessed in a short amount of time) is likely in the Data Location Cache. In some scenarios, the cache may be optional. In particular, when the application can find the location of data resources without complex processing (e.g., by just applying *consistent hashing* on the fields of a service request), the *Admission Control* block can perform the same computation and find the location of the resources. In general, it is useful to still have a distributed cache if the application needs to access a large number of resources per service request, and when the computation of data location is expensive. Otherwise, if service requests involve only a few resources and there are no hot-spot resources (e.g., as in the case of an IMS scenario discussed in Section 4), it can be advantageous to compute the location directly, avoiding using the cache.

3.2. Distributed Capacity Monitoring

The *Distributed Capacity Monitoring* component is deployed within all datastore nodes, and it is in charge of dynamically estimating their available capacity. This block collects information about resource utilization from the guest OS or the hypervisor of the nodes. We focus the discussion on the case where each monitoring block estimates the capacity of a storage node in terms of its *CPU utilization* (i.e., in terms of percentage of busy CPU cycles per unit of time), since the CPU is often the resource most prone to become a bottleneck [36]. In addition to CPU utilization, the proposed solution can be easily generalized to be applied to memory, network, and disk bandwidth utilization.

This block periodically estimates the available capacity, in terms of the number of requests that the node can serve in the next time window. The time window is meant to be short (e.g., in the order of few seconds), in order to quickly adapt to variations of the load of the datastore node. In addition to CPU utilization, the *Capacity Monitoring* block tracks the number of data requests that have been served by the storage node in the last time window, which is recorded by the *Admission Control* block when a service request is accepted. The available capacity is estimated as follows:

$$\begin{matrix}\text{available}\\ \text{capacity}\end{matrix} = \frac{\#\ \text{data requests}}{\texttt{CPU}_{\texttt{used}}\%} \cdot \texttt{CPU}_{\text{reference}}\% \quad . \qquad (1)$$

The first factor estimates the cost of an individual data request, in terms of CPU cycles, by dividing the number of data requests in the last time window with the average CPU utilization during the same period. This formula is a valid approximation of available capacity when `CPU_used` is lower than `CPU_reference`. However, when this condition does not hold, it has the effect to progressively reduce the number of data requests to be accepted by a factor `CPU_reference / CPU_used` until `CPU_used` value becomes less than `CPU_reference`. This approximation is simple but still accurate in our context (modern multi-tier systems based on NoSQL datastores), since the complexity of an individual data request is relatively low. For older types of storage systems (e.g., based on a traditional SQL DBMS), the cost would depend on the SQL queries performed by the application, and it should be estimated using a more complex cost model [37]. Since we focus this work on modern datastores, we leave out of scope the analysis of other cost models. The second factor in the equation represents the *reference CPU budget*, beyond which the datastore node is considered saturated. This value represents a "factor of safety" for CPU utilization, within which the datastore node is designed to perform well (e.g., without performance disruptions), while leaving a small amount of residual CPU bandwidth to handle occasional load fluctuations. We base our algorithm around a reference value, since setting a reference is a common practice among system administrators, e.g., for monitoring and troubleshooting purposes. The Algorithm 1 executes periodically to update the capacity budget of the datastore nodes, according to the request rate and CPU utilization measured during the last period.

ALGORITHM 1: Capacity Monitoring

```
// Executes periodically
begin
    // Get CPU utilization and number
    // of data requests since
    // the previous update
    $CPU_i = \text{get\_CPU\_utilization}(node_i)$
    $requests_i = \text{get\_served\_data\_requests}(node_i)$

    // Compute available capacity (Eq. (1))
    $C(i) =$
      $\text{compute\_capacity}(node_i, CPU_i, requests_i)$
    $\text{update\_datastore\_capacity\_budget}(node_i,\ C(i))$
end
```

3.3. Distributed Admission Control

Figure 3 shows the internal organization of the *Distributed Admission Control.* When a service request is received, a *Data Location Discovery* procedure is performed before the request is forwarded to the application for processing. The discovery procedure identifies the set of data resources needed by the service request, and the datastore nodes where these resources are located. The *Distributed Admission Control* uses this list for deciding whether to admit the request, according to the available capacity of datastore nodes.

The implementation of this component depends on the specific application, and it is meant to be tailored by the application programmer. The *Data Location Discovery* parses service requests by looking for information that uniquely identifies the data needed by service requests, such as the user identity, the session identifier, the resources requested by the user (e.g., multimedia content), and the type of operation to be performed on the resource. This information is then used to query the *Distributed Memory* to find the location of resources in the datastore tier. The main assumption of the proposed solution is that service requests hold information to establish the mapping with datastore nodes. This assumption holds in practice for many applications: since application nodes are stateless, the service request message includes all the information needed by the application logic to access the datastore. We will further discuss this aspect in the context of two case studies (Sec. 5 and 4).

If the service request involves resources that may have already been accessed in the past, the *Data Location Discovery* block checks if there is an entry for that service request in the *Distributed Memory*, and retrieves information on the resources from there. If an entry does not exist yet (for example, the service request comes from a new user), the *Data Location Discovery* block uses the information extracted from the request to find the location of the required resources in the datastore tier, and updates the cache.

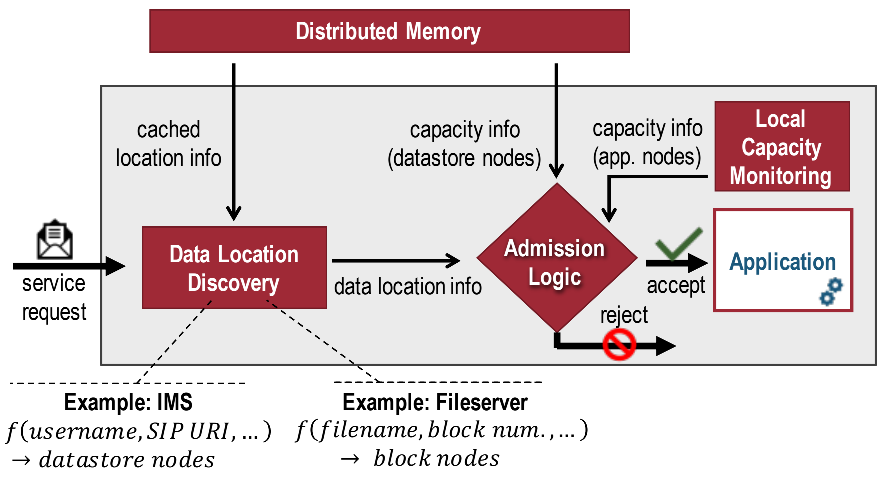

Distributed Memory
cached location info
capacity info (datastore nodes)
capacity info (app. nodes)
Local Capacity Monitoring
service request
Data Location Discovery
data location info
Admission Logic
accept
reject
Application
Example: IMS
$f(username, SIP\ URI, \dots) \rightarrow datastore\ nodes$
Example: Fileserver
$f(filename, block\ num., \dots) \rightarrow block\ nodes$

Figure 3: The Admission Control process.

For example, the *Data Location Discovery* block can compute the same hash function that is computed by the consistent hashing mechanism (Sec. 2) to resolve the resource location. In other cases, the *Data Location Discovery* block obtains the location of a resource by retrieving resource metadata. We refer to this computation as the *location function*, which maps the information from a request (i.e., the "key" of an entry in the datastore) with the resource location. For instance, in the case of a File Server, the resource is represented by a file block; the key is represented by the combination of the filename and of the numeric identifier of the block; and the output of the function is the datastore node with the block. In the case of a Multimedia Server, the resource is represented by a record with user information; the key is represented by a combination of the username and the SIP URI; and the output is the datastore node with the record. Since both the key extraction from the service request and the location function are simple operations, their overhead is expected to be small.

Alg. 2 details the *Distributed Admission Control* process. The process first checks if local application node is not overloaded. Then, it gets the location array $L(1..n)$ from the *Data Location Discovery* procedure: the i-th component of this array represents the number of data requests that will be directed to the storage node i. Moreover, the *Admission Control* retrieves the available capacity array $C(1..n)$, where n is the total number of datastore nodes: the i-th element of this array represents the number of data requests that the datastore node i can accept in the current time window without saturating its capacity. This node capacity status is updated on the *Distributed Memory* by the *Capacity Monitoring* block. The algorithm compares, for each storage node i, the available capacity of the node with the number of data requests for the node. When $L(i) > 0$, there is at least one resource on the i-th data node required to complete the current service request. If there is at least one storage node in which the residual capacity is not sufficient to process the data requests (i.e., $C(i) - L(i) < 0$), the algorithm rejects the service request. Otherwise, if the service request is accepted, the algorithm discounts the number of data requests towards node i from its available capacity budget, and updates the budget on the *Distributed Memory*.

4. IP Multimedia Subsystem Case Study

We here consider a case study from the telecom domain, where overloads are a recurrent issue, and where high performance and availability are key concerns [38, 9, 10]. A current trend (*Network Function Virtualization*, NFV) in this domain is the migration of services from traditional hardware appliances to *softwarized* ones, and to run them on cloud computing infrastructures [39]. The goal of NFV is to reduce costs and improve manageability, while achieving the same, or even better, reliability and performance of traditional hardware appliances. The IP Multimedia Subsystem (IMS), i.e., an architectural framework for delivering multimedia content over internet, is one of the most relevant

ALGORITHM 2: Admission Control

```
// On arrival of a service request
begin
    $C_{local}$ = get_local_capacity_budget()
    if $C_{local}$ = 0 then
        REJECT the request
    end
    $M$ = get_metadata(app_request)
    $L$ = get_data_location($M$)
    foreach datastore node $n_i$ in $L$ do
        // Check if there is enough capacity
        $C(i)$ = get_datastore_capacity_budget($n_i$)
        if $C(i) - L(i) < 0$ then
            REJECT the request
        end
    end
    // Update capacity budgets
    foreach datastore node $n_i$ in $L$ do
        update_datastore_capacity_budget($n_i$,
          $C(i) - L(i)$)
    end
    update_local_capacity_budget($C_{local} - 1$)
    ACCEPT the request
end
```

use cases for NFV.

We analyze the impact of unbalanced overloads on the open-source Clearwater IMS [40], deployed on an experimental testbed with 100+ virtual machines. Clearwater is an NFV-oriented implementation of IMS (Figure 4): all components are horizontally scalable using simple, stateless load-balancing; the various components are decoupled, and interact through standard interfaces through connection pooling; long-lived state is stored in datastore nodes using NoSQL technologies. In particular, the IMS uses the Memcached datastore, which is the most commonly used among popular applications that need to manage data over thousands of datastore nodes [41, 42, 43, 44, 45]. The datastore holds data of IMS users (*subscribers*) related to authentication and billing information, and requests are balanced across the tier through consistent hashing. In this architecture, unbalanced overloads can be caused by resource hogs and by software and configuration issues.

To assess our proposed solution (labeled as "*data-dependency-aware*"), we also consider a baseline representative of existing overload control solutions (labeled as "*non-data-dependency-aware*").

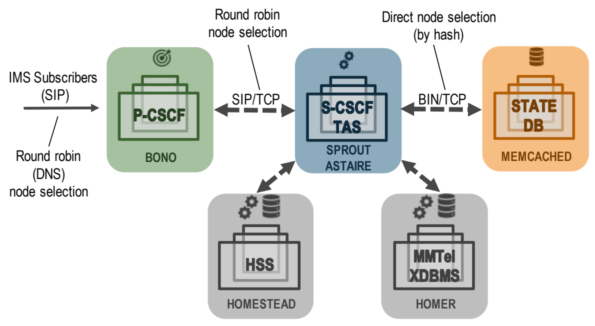


Figure 4: Overview of the Clearwater IMS.

As baseline, we consider the overload control mechanism included in the Clearwater IMS [46, 47]. This algorithm uses a *token bucket* to control the rate of requests that a node can process. The token replacement rate is tuned by computing the smoothed average latency of processed requests, and by comparing this metric with a configured latency target. This algorithm is an evolution of the one proposed by Welsh and Culler [32], and representative of latency-based overload control typically adopted in distributed systems.

4.1. Integration of the overload control solution

To apply the overload control solution, we deploy the *Capacity Monitoring* and the *Admission Control* respectively on the *Memcached* and *Sprout* nodes. The *Distributed Memory* runs on a dedicated set of standalone nodes in order to analyze its overhead (Sec. 6). The *Data Location Discovery* block implements the procedure in Figure 5. It extracts from the incoming SIP messages the user identity (e.g., `50012345@example.com`) and, in case of an INVITE message, the identity of the callee (e.g., `5001244@example.com`). On each user request, the *Sprout* node accesses the information about the user session in JSON format, by performing a query on Memcached, using the key `reg\\user_identity` (e.g., `reg\\50012345@example.com`).

The *Data Location Discovery* procedure looks-up the *Memcached* node by computing a hash function on the key for the user's data (e.g., `MD5(reg\\50012345@example.com)`). In this case study, the *Data Location Cache* is optional, as the hash function can be computed on every request with a small overhead. Since the data location can be determined solely from the request, we use the

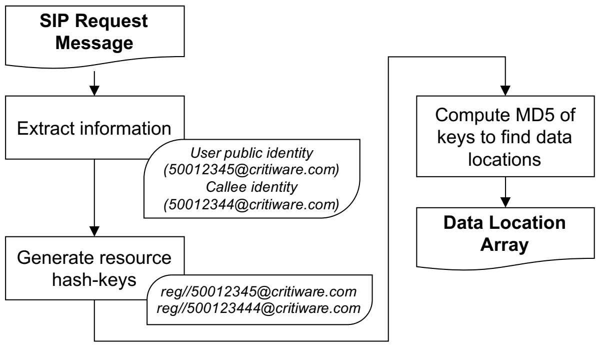

SIP Request Message
Extract information
User public identity (50012345@critiware.com)
Callee identity (50012344@critiware.com)
Generate resource hash-keys
reg//50012345@critiware.com
reg//500123444@critiware.com
Compute MD5 of keys to find data locations
Data Location Array

Figure 5: Data Location Discovery for the IMS.

Distributed Memory to only store capacity information about the datastore nodes.

After a `REGISTER` or an `INVITE` message, the IMS and the user agent generate a flow of messages, by following a sequence specified by the SIP protocol (e.g., `INVITE` - `100 Trying` - `180 Ringing` - `200 OK` - `ACK`). In all these messages, the server will request the same key, and thus will query the same datastore node. Thus, before admitting a new user session, on the first `INVITE` message we check whether there is enough capacity to satisfy all of the subsequent messages in the sequence. The *Admission Control* algorithm takes into account the type of the request, and performs throttling on the first REGISTER and the first INVITE message of a SIP session. This approach gives priority to SIP sessions that are already established, to prevent user-perceived errors in the middle of a SIP session.

4.2. Experimental setup

The experimental testbed consists of eight host machines SUPERMICRO (high density), equipped with two 8-Core 3.3Ghz Intel XEON CPUs (32 logic cores in total), 128GB RAM, two 500GB SATA HDD, four 1-Gbps Intel Ethernet NICs, and a NetApp Network Storage Array with 32TB of storage space and 4GB of SSD cache. The hosts are connected to three 1-Gbps Ethernet network switches, respectively for management, storage and VM network traffic. The testbed is managed with OpenStack Mitaka and the VMware ESXi 6.0 hypervisor. In order to reproduce unbalanced overload conditions, we adopted a large IMS deployment, which includes 50 nodes in the *Sprout* application tier, 50 nodes in the *Memcached* datastore tier, and 10 nodes in the *Bono* front-end tier. We configured the number of nodes for the other components of the IMS proportionally to the capacity of the *Sprout*, *Bono*, and *Memcached* nodes, such that to have an average CPU utilization in each component of 80% with no request failures. We deployed a cluster of 10 VMs for workload generation, using *SIPp* [48] to generate a SIP workload for the IMS system. SIPp generates message flows between subscribers and the IMS according to the SIP protocol, and it is also by the Clearwater project to perform tests on the IMS. The SIPp configuration used in our experiments is available online [49].

In order to perform a conservative evaluation of the overload control solution, we simulate a worst-case overload scenario, where a higher number of subscribers and requests suddenly enter the system at the highest possible rate. As this work has been conducted in the context of an R&D cooperation with a commercial vendor of NFV products and services, we tuned the behavior of the subscribers (e.g., the rate of busy-hour call attempts) according to the experience of the company with overload conditions [25]. In this workload, every subscriber registers and periodically renews the registration every minute on average. After a successful registration, a subscriber attempts to set up a call with another subscriber (with 16% of probability), or remains idle until the next registration renewal (with 84% of probability). The call hold time is, by default, 60 seconds. Thus, the scenario reproduces 60 *Busy Hour Register Attempts* (BHRA) per user and 5 *Busy Hour Call Attempts* (BHCA) per user. Our deployment can handle up to 110k subscribers without exhibiting any failure, corresponding to 1,833 REGISTER/s and 153 INVITE/s on average, with an average CPU utilization of 80%.

We adopted the following experimental plan. We consider different volumes of workload, by varying the number of subscribers. The workload volume range from normal conditions (request rate at 70% and 100% the engineered capacity, with 80k and 110k subscribers) to overloading ones (respectively, request rate at 4, 10 and 100 times the engineered capacity of the system, up to 11M subscribers). The workload surges assess the ability of overload control to discard requests in excess, and to ensure a high throughput of served requests. Moreover, we make the overload condition to be unbalanced in the datastore tier, by injecting resource contention in 5 out of 50 datastore nodes, in order to assess whether overload control can achieve high resource utilization of the non-overloaded nodes. All experiments are divided in 3 phases, as follows:

1. **Initial ramp-up phase**: We introduce new subscribers in the system, up to the engineered capacity, and wait for a steady state. We use this condition as a starting point for the evaluation.

2. **Workload surge**: We add more subscribers to generate a workload surge (up to 100 times the engineered capacity) to cause an overload condition.

3. **Hog injection**: We simulate an unbalanced overload condition in the datastore nodes (e.g., reduced capacity due to *hogs* on the same machine, or bad configuration), by injecting delays in Memcached's request handling.

The IMS is a performance-critical application, which needs to assure a low response time (*latency*) even under stressful conditions. We consider typical latency constraints, where registration requests and call-setup requests should be served within 100ms and 250ms at most, respectively [38, 29]. In order to evaluate both the service rate and latency achieved by the IMS, we evaluate the *useful throughput* of served requests, i.e., the number of requests per second that are correctly processed within latency constraints. Requests that exceed latency constraints are treated as errors, thus lowering the useful throughput.

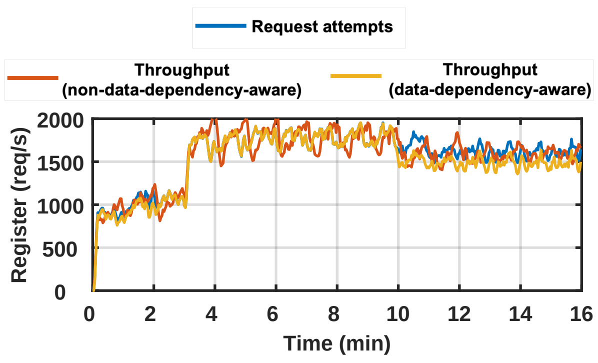
4.3. Experimental results

We first present in detail a case of workload within the engineered capacity of the system (1x load), while injecting an unbalanced load condition in the datastore tier, in order to analyze the performance overhead of overload control, and the effects of resource contention. Then, we present in detail a representative experiment workload above the engineered capacity (10x load), to analyze the combined effects of workload surge of subscribers and unbalanced overloads in the datastore tier. Finally, we analyze the IMS performance over all of the experiments.

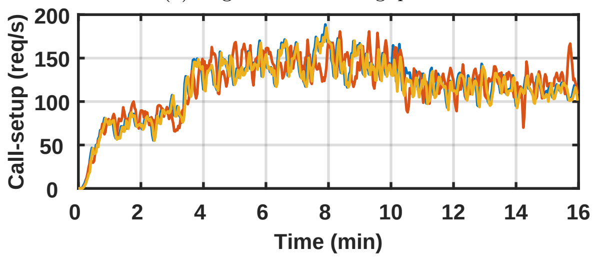
Figures 6a and 6b show the performance of the IMS for registration and call-setup requests, with a workload at the engineered level (1x), by generating 110k subscribers. During the steady state (starting at minute 4), after all of the subscribers performed an initial registration, the system is able to process 1800 registrations/s and 150 call-setup/s on average. The throughput of the overload control solutions are matching, and both close to the rate of the incoming requests from the clients. The proposed overload control solution incurs a negligible overhead, since it only needs to perform a lightweight parsing of few bytes of the requests, to extract the request type and user identities, and to compute the hash function. Thus, the proposed overload control solution does not interfere with the performance of the system under normal conditions.

Later during the experiment at minute 10, when the hog is injected, the registration throughput decreases by 12% for both overload control solutions, but with different behaviors. The proposed overload control solution selects requests based on their data dependencies, and rejects them before they enter the system, thus avoiding that these requests could reach the overloaded datastore nodes. Instead, with non-data-dependency-aware overload control, requests in excess are still admitted in the system, and fail later when the requests hit the (overloaded) datastore nodes. This second behavior is undesirable, since it exacerbates the overload condition of the datastore nodes (increasing the likelihood of software failures), wastes resources (some sessions will partially processed and eventually fail), and causes long delays experienced by the clients (i.e., they only notice request failures after a timeout). More insights on this effect can be observed in Figure 6c, which shows the CPU consumption of an overloaded datastore node, when a CPU hog injected after minute 10. In the case of non-data-dependency-aware overload control, the CPU utilization saturates to 100% as an effect of requests in excess that are still admitted. The data-dependency-aware throttling prevents these requests in excess from reaching the datastore node, and stabilizes CPU utilization below 80%.

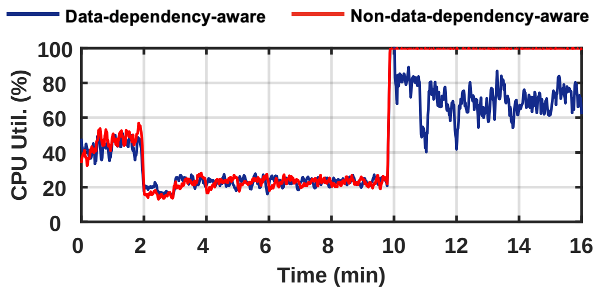
Figures 7a and 7b show the performance of the IMS when the workload exceeds the engineered capacity by 4 times. After the ramp-up phase, we generate a workload surge at minute 4, by introducing up to 440k subscribers. With non-data-dependency-aware overload control, the registration throughput significantly decreases, down to 105 registrations per second on average. Moreover, the call-setup throughput becomes even lower, since even the registered users are able to initiate calls due to the high resource contention. As pointed out in Figure 7c (between min 4-10), the CPU utilization of datastore nodes is highly variable, as non-data-dependency-aware overload control only indirectly adapts to the load of the datastore tier, by



(a) Registration Throughput

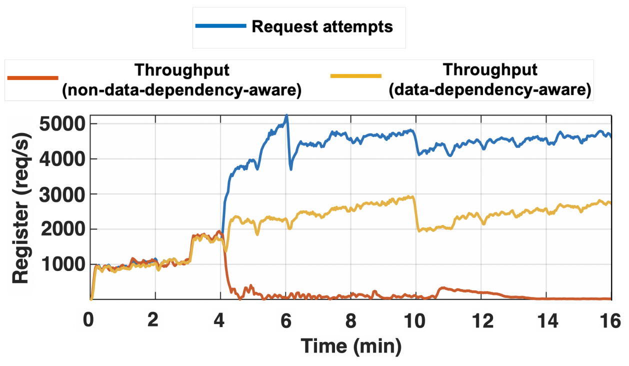


(b) Call-setup Throughput

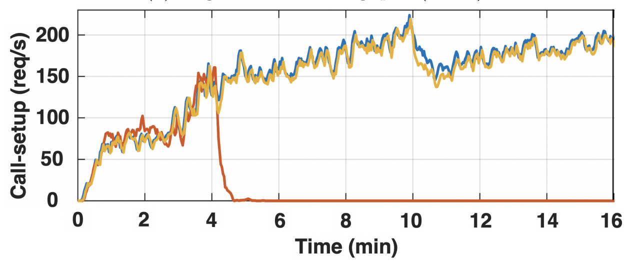


(c) Memcached CPU consumption

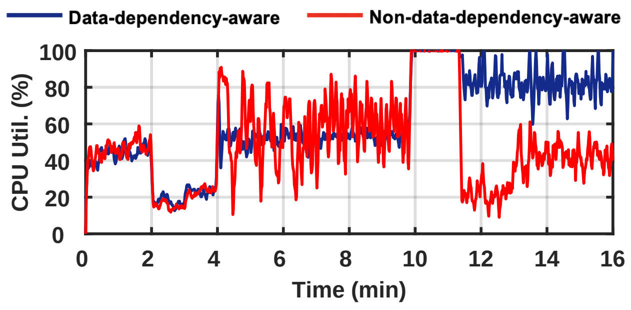
Figure 6: IMS performance at the engineered capacity (1x).



(a) Registration Throughput (400%)



(b) Call-setup Throughput (400%)



(c) Memcached CPU consumption

Figure 7: IMS performance at 4x the engineered capacity.

averaging request delays over a time window. As a result, too many requests are admitted in the system, which eventually lead to the failure of several application nodes due to out-of-memory kills by the OS.

With the proposed data-dependency-aware overload control solution, the throughput is always above the engineered throughput, for both registration and call-setup operations. Registrations in excess (i.e., the difference between request attempts and throughput in Figure 7a) are timely rejected, without experiencing failures of the application nodes, and with a stable CPU utilization (Figure 7c). The admitted subscribers are able to make call-setup requests at full capacity (i.e., there is no gap between request attempts and throughput in Figure 7b), thus achieving a high quality of service for the admitted subscribers. Interestingly, during the workload surge, the registration throughput is even higher than the engineered level (over 2000 regs/s). This behavior is caused by the lower number of datastore accesses (2) made by re-registration requests, compared to datastore accesses (6) made by new registrations. As subscribers get admitted and lower their datastore accesses, the system opportunistically admits some of the new subscribers in excess, thus gradually increasing the overall amount of registrations.

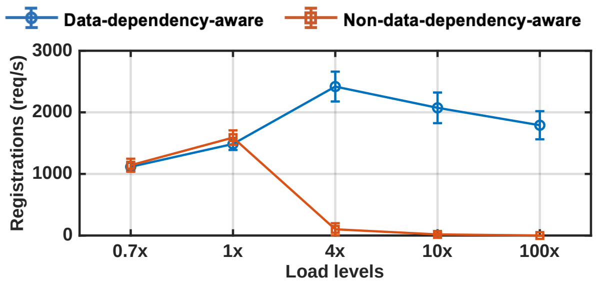
During the injection of resource hogs, the sessions that are currently served by the injected nodes are slowed down, causing a CPU saturation for about 1 minute (after min 10, in Figure 7c), with both overload control solutions. When these sessions end, the data-dependency-aware solution is able to keep the CPU utilization around 80%, since it avoids accepting any new session that accesses the overloaded datastore nodes. Instead, in the non-data-dependency-aware case, the application nodes keep submitting new requests to the overloaded datastore nodes, which are enqueued and cause the exhaustion of the socket pool. This behavior, in turn, causes failure in application nodes.

Even during the injection of a hog, overload control preserves the registration throughput for already-registered sessions (Figure 7a), which is still above the engineered level. Sessions are not admitted if any of its data requests access any of the overloaded datastore nodes. In this process, overload control also prevents some data requests to non-overloaded datastore nodes. Thus, datastore nodes become less loaded, so other sessions (i.e., the ones that do not use overloaded nodes) are gradually admitted in place of the rejected ones.

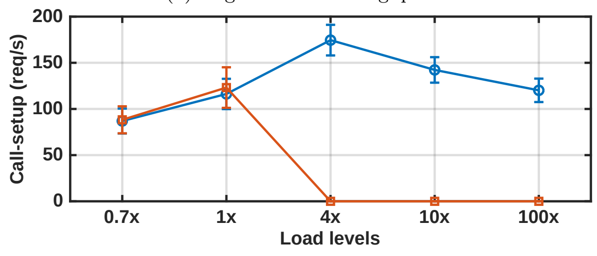
Figure 8 summarizes the performance of the IMS (average and standard deviation of the throughput for registrations and call-setups) under hog injection, for all workloads, and for both the overload control solutions. When the load is within the engineered capacity (0.7x and 1x), there are no significant differences between the two overload control solutions. However, as discussed before for the 1x case (Figure 6), data-dependency-aware overload control prevents unnecessary resource waste and delays, by rejecting sessions before they enter the system. Under higher load conditions (4x, 10x, and 100x), the performance degrades, and the IMS components eventually experience software crashes due to resource exhaustion. When we introduce the data-dependency-aware overload control, the IMS does not experience crashes, and reaches a stable throughput above 90% of the engineered capacity.

5. Distributed Fileserver Case Study

We here analyze the overload control solution in the context of a case study on a *distributed fileserver* for cloud storage, which has high performance and availability requirements, and is prone to hot-spots [50, 51]. This case study is based on a proprietary system from our industrial project partner, where we experiment with unbalanced overload scenarios that the company experienced in this context. This system includes three tiers (Figure 9): a frontend



(a) Registration Throughput



(b) Call-setup Throughput

Figure 8: Overview of results on the IMS.

tier, based on *HAproxy*, which performs load balancing; an application tier, composed of web application nodes; and the datastore tier, composed of *Memcached* nodes.

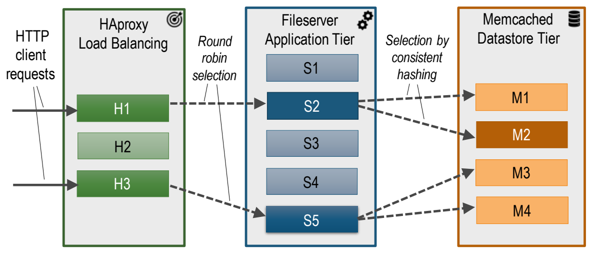


Figure 9: Overview of the distributed fileserver case study.

The user can perform 4 types of operations on the system: (1) registration, (2) file upload, (3) file download, and (4) de-registration. Clients select an instance of the HAproxy by querying a DNS server (*bind9* in our setup). The application tier is stateless: no session state is stored in the web application nodes. For example, the same client can send a registration request to the server S1, and an upload request to the server S2. The web application stores the data in the Memcached key-value store cluster. These services are implemented by a C application, based on *libevent* library for asynchronous communication with clients, and on *libmemcached* for communication with the datastore.

The registration is a "set" operation that stores user account information on a datastore node (such as the username and the last access time) under a specific key, while the de-registration is a "delete" operation that removes the key-value pair from the server. To make an upload request, the client sends a file through an HTTP POST request message that includes the "username", the "filename" and the file contents. Then, the application divides the file contents into chunks of equal size (1MB), and stores the chunks into data nodes. The application uses the string key `"username$filename$i"` to identify the *i*-th chunk. Then, sequentially for each chunk *i*, the application computes the hash function `MD5("username$filename$i)` to identify the location of a datastore node where to save the chunk. Finally, the application uses the key `"username$filename"` to store the file size obtained from the `Content-Length` HTTP header. The upload operation uses the *setMulti* function to store multiple key-value pairs on datastore nodes. For download requests, the application uses the same strategy. It extracts the "username" and the "filename" from the request message. Then, it gets the current file length by querying the key `"username$filename"`, and computes the hash `MD5("username$filename$i")` for every chunk *i*. The download operation uses the *getMulti* function to concurrently retrieve multiple chunks from datastore nodes.

5.1. Integration of the overload control solution

To apply the overload control solution, we deploy the *Capacity Monitoring* and *Admission Control* respectively on the datastore and the application nodes. We deployed the *Distributed Memory* in a set of dedicated Memcached nodes, in order to assess its overhead (further discussed in Sec. 6).

For this case study, the *Data Location Discovery* procedure (Figure 10) parses the HTTP Request message, to extract information about the request type, the username, the filename, and the content-length. Then, it builds the same group of hash keys generated by the application (e.g., `user8`, `user8$file.txt`, `user8$file.txt$chunk1`, ...). Using the request identifier (i.e., `username` for the register and the unregister operations, and `username$filename` for the upload and the download operations), the admission control performs a cache lookup to identify the nodes needed to perform the operation. In case of a cache miss, the admission control finds these locations by computing the hash function, and creates a new entry in the cache. Once the locations have been computed, the admission control algorithm evaluates the location array L, where the $i-th$ element represents the number of file blocks to be accessed on the datastore node i.

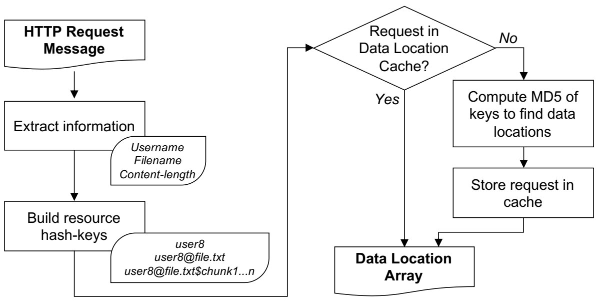


Figure 10: Data Location Discovery for the fileserver.

5.2. Experimental setup

The experimental testbed is based on the same hardware and software as the previous case study (Sec. 4). In order to reproduce representative unbalanced overload conditions, we set up a testbed with a large number of nodes. We use 50 independent VMs for the application tier, and a further 50 VMs for the datastore tier. Service requests are load-balanced among 10 VMs by HAProxy in the front-end tier.

We performed a conservative evaluation of the proposed solution, by simulating a worst-case overload scenario, where hot-spot resources are suddenly requested at the highest possible rate. The workload is generated by Apache JMeter in a distributed configuration. We deployed 10 additional VMs to submit requests to the system, and a controller VM to set up the experiment and collect performance data, including application latency, throughput, and service failures. We tuned the workload generators following the experience of our industrial partner. The workload runs a set of clients, which register on the system and perform several uploads and downloads of files with a size between 8KB and 4MB, and random contents. In this configuration, the system can handle up to 700 concurrent users with no failures, achieving an average throughput of 175 uploads/s and 175 downloads/s.

We adopt an experimental plan with different unbalanced overload conditions in the datastore tier.

Initially, we apply a workload with a balanced request mix at a rate within the engineered capacity of the system, in which each user requests its own files. Then, we apply a skewed workload, by introducing additional clients that repeatedly access a shared set of *hot-spot* files, at a rate 4, 10, and 100 times the engineered capacity. The files are located across a set of 7 to-be-overloaded nodes out of 50 datastore nodes. We vary the number of these clients between 0 (i.e., 1x balanced workload) and $70K$ (100x skewed workload on the hot-spots). All experiments are divided into 3 phases, as follows:

1. **Initial ramp-up phase**: We introduce new clients in the system, up to its engineered capacity, and wait for a steady state. We use this condition as starting point for the evaluation.

2. **Hot-spot generation**: We increase the workload by introducing additional clients that access shared files, to generate hot-spots in the datastore.

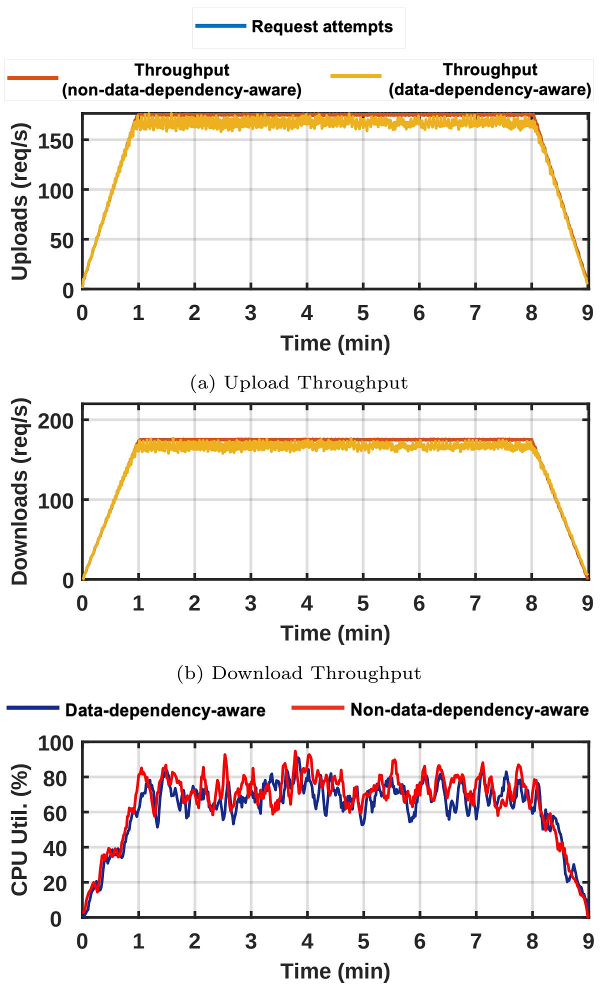
3. **Final ramp-down phase**: We gradually reduce the workload until we remove the effects of the hot-spot.

In this case study, we again evaluate the *useful throughput* of the system. The application treats as errors the requests that experience latency higher than 200ms. Thus, any significant latency increase will show up as lower throughput.

5.3. Experimental results

We first discuss the case of normal conditions, with a balanced load at the engineered capacity. Then, we discuss the effect of hot-spots, with reference to the case of 10x skewed workload. Finally, we compare the overall throughput across all the experiments (1x, 4x, 10x, 100x workload on hot-spots).

The experiment with balanced load within the engineered capacity (1x) showed that our solution has no side effects when the system is under normal conditions (Figures 11a and 11b), since both the upload and download throughput of the system are still at the engineered capacity, and no failures are experienced. The CPU utilization on the datastore nodes is also comparable for the two solutions Figure 11c. When performing data-dependency-aware control, we did not observe any latency violation



(a) Upload Throughput

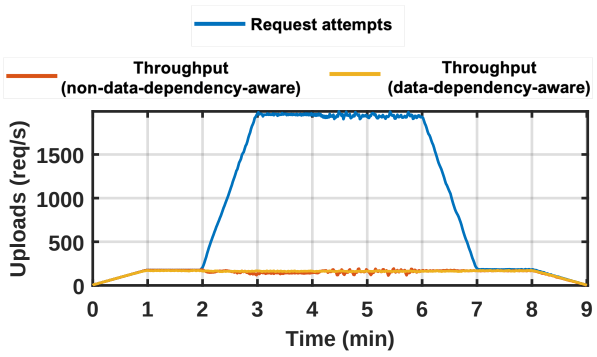
(b) Download Throughput

(c) Memcached CPU consumption

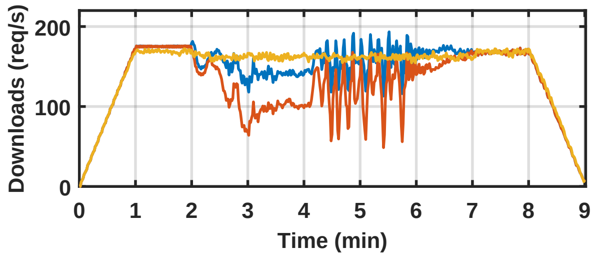
Figure 11: Fileserver performance at engineered capacity (1x).

under normal conditions, as the *Data Location Discovery* uses caching to quickly look up the locations of data dependencies.

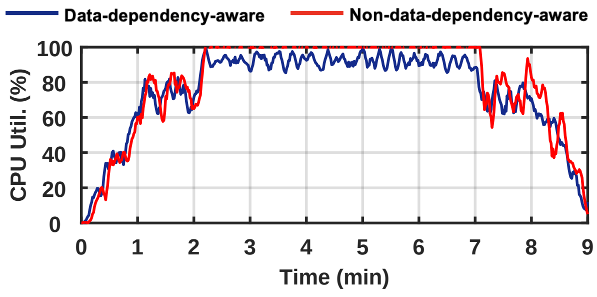
In the experiments with skewed workload on hot-spots, we observed a significant degradation in the case of the baseline approach, both in terms of upload and download throughput (between min 2-7 in Figure 12a and 12b, with reference to 10x skewed workload). The throughput of download operations is lower than upload operations, since the two operations are not independent, and the slow-down and failure of upload operations cause a decrease of download requests. The requests for hot-spot files have an impact on all users in the system, since the system tries to process all requests at the same time, but fails at completing most of them. This behavior



(a) Upload Throughput



(b) Download Throughput

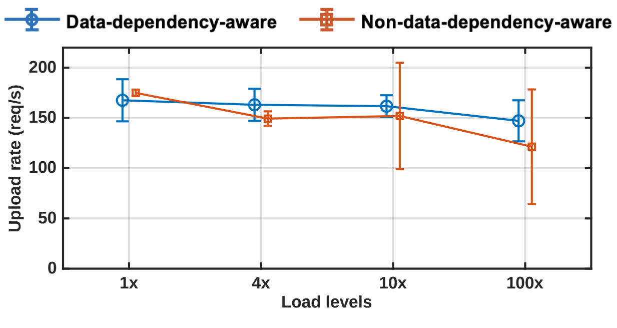


(c) Memcached CPU consumption

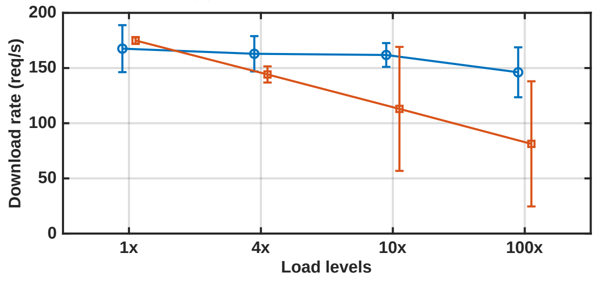
Figure 12: Fileserver performance at 10x engineered capacity.

can be observed in the CPU utilization of datastore nodes with hot-spot resources (Figure 12c). In the baseline case, CPU utilization saturates to 100%.

Instead, with data-dependency-aware overload control, both the upload and download throughput only experience a moderate decrease of the throughput (Figure 12a and 12b). The *Data Location Discovery* is able to keep up with the high rate of requests for hot-spot resources, as the location of these resources is quickly recorded in its cache. The overload control solution keeps the CPU utilization at 85% (Figure 12c), which is the target CPU utilization that we configured in the *Admission Control* to prevent excessive resource contention. Finally, after the hot-spot phase, the CPU utilization reduces as expected (min 7-9).



(a) Upload Throughput



(b) Download Throughput

Figure 13: Overview of results on the distributed fileserver.

All experiments are summarized in Figure 13. In the baseline case (non-data-dependency-aware), the fileserver exhibits a noticeable throughput degradation. In the worst case (100x overload), the throughput reduces to about one-half on average for uploads, and about one-third on average for downloads. Instead, with data-dependency-aware overload control, the throughput is above 90% of the engineered throughput, even in the worst case of the 100x overload condition, both for uploads and downloads. By preventing the admission of requests that are overloading hot-spot datastore nodes, the overload control solution leaves the system with enough available capacity to serve all the admitted requests, even during a high workload surge.

6. Overhead and scalability

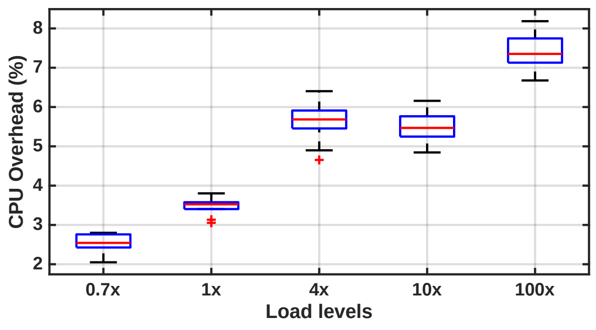
Since our solution is meant to be deployed in multi-tier systems, with a large number of nodes in each tier (e.g., up to 10k nodes in production systems at our industrial partners), we optimized this component to achieve high scalability and low overhead.

The proposed solution deploys an agent on each application node to perform data-dependency-aware overload control. This agent has a small memory footprint, with less than 10 MB even in the 100x overload case. The agent consumes some CPU for request inspection and admission, depending on the volume of incoming requests. In our experiments, the CPU overhead was small, and less than 8% in the worst-case 100x load (Figure 14).

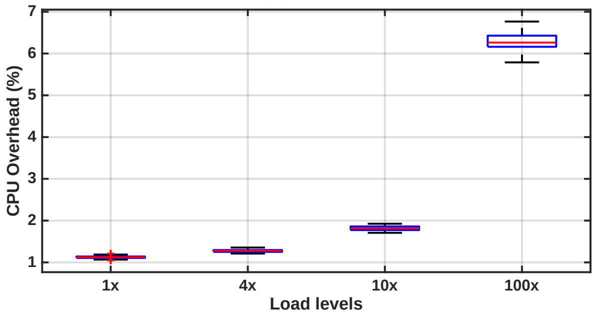
The computational cost of the overload control algorithms is limited. The Algorithm 1 executes periodically (e.g., every few seconds) to get information about the load of datastore nodes, and perform simple computations, similarly to monitoring solution that are adopted by cloud system administrators. Thus, it has a constant computational cost with respect to the number of client requests, and scales linearly with the number of nodes in the datastore tier.

More importantly, the communication cost (i.e., number of exchanged messages with other nodes) of the Algorithm 1 is also small and scales linearly both with the number of nodes in the application tier, and of nodes in the datastore tier. We consolidate the monitored data into the *Distributed Memory* stored in an additional NoSQL datastore, which runs on a small set of nodes to avoid becoming a performance bottleneck, and which transparently manages the distribution of key-value pairs, with low memory consumption. Moreover, the *Distributed Memory* keeps at a minimum the communication latency by using a NoSQL datastore, which is lightweight and schemaless, and by establishing a fixed pool of persistent connections with each node. As discussed in experimental evaluation of Sec. 4 and 5, the impact on the tail latency is small enough to prevent violations of latency constraints. Since there is no direct communication between nodes in the application and datastore tiers, the number of connections only grows linearly with the number of nodes in the tiers.

The Algorithm 2 also has a limited computational and communication cost. The algorithm is executed on every incoming serving request. The algorithm finds the location of datastore nodes by extracting metadata from the client's request message. The algorithm only iterates over a small subset of datastore nodes, i.e., the ones that hold the data needed by the incoming service request. Moreover, the algorithm only communicates with the *Distributed Memory*, to retrieve information about the current available capacity of these nodes. Therefore, the communication cost does not increase when scaling up the datastore tier.



(a) IMS



(b) Distributed fileserver

Figure 14: CPU utilization of the overload control agent.

6.1. Sizing the Distributed Memory

We here analyze the *Distributed Memory* component at increasing scale, in order to quantify the cost of deploying the proposed solution, in terms of additional nodes.

Let T_A be the *engineered throughput of a single application node*; N_A and N_C respectively the *number of application nodes* and of *distributed memory nodes*, respectively; and $\overline{R}$ the *average number of accesses to the Distributed Memory per service request*. Each application node has a pool of p connections to each of the N_C nodes of the Distributed Memory, to avoid opening TCP connections on-demand. Therefore, the total number P of connections is constant ($P = N_A \cdot p$).

The maximum number of requests towards the Distributed Memory is limited by the engineered capacity of the application tier. If requests exceed the capacity of the application node, they are rejected without any further inspection (the *local capacity* in Alg. 2). Thus, the maximum number of requests inspected by an overload control agent is equal to the engineered capacity of the application node. We denote as $T_C^{(P)}$ the *maximum throughput of an individual Distributed Memory node* at concurrency level P. This throughput can be experimentally obtained by performing a simple *capacity test*, using a synthetic workload generator to generate P concurrent connections and to measure the throughput, as discussed in the next subsection.

The Distributed Memory as a whole can handle at most $N_C * T_C^{(P)}$ requests. This capacity must match the maximum number of requests that come from the application tier, which is $N_A * T_A * \overline{R}$ when the workload reaches the engineered capacity of the system, that is, $N_A T_A \overline{R} = N_C T_C^{(P)}$. Therefore, the *number of Distributed Memory nodes* needed to sustain the overload control algorithm is given by:

$$N_C = \left\lceil \frac{N_A T_A \overline{R}}{T_C^{(P)}} \right\rceil \qquad (2)$$

6.2. Example: scaling the solution up to 10K nodes

We performed a capacity test to estimate the maximum throughput of a *Distributed Memory* node. We used the *memtier-benchmark* tool to generate synthetic workload for *Memcached*. Figure 15 shows the throughput $T_C^{(P)}$ for a number of concurrent connections P ranging from 50 to 50k.

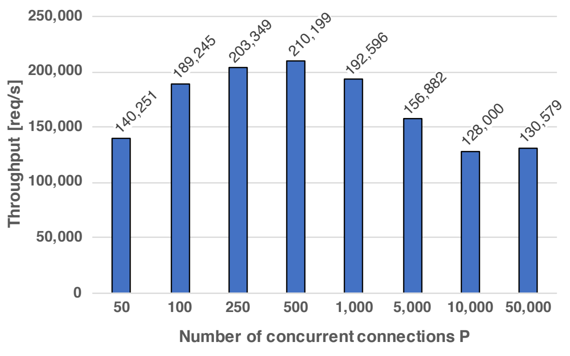


Figure 15: Capacity test of a Distributed Memory node.

Using Eq. 2 and the results from the capacity test, we evaluate the number of Memcached nodes needed to apply our solution in a system with 10k application nodes.

A) Our testbed configuration (up to 50 application nodes). The number of application nodes is $N_A = 50$. The average number of requests to the Distributed Memory per service request is $\overline{R} = 4$. The engineered throughput of a single application node is $T_A = 40$ req/s at the steady state (the whole throughput is $2,000$ req/s). The connection pool size in each application node is $p = 5$ connections. The concurrency level of each Memcached server is $P = N_A \cdot p = 250$ connections. Each server is configured with 1 vCPU and 4GB RAM. The throughput of a single Memcached server at this concurrency level is $T_C^{(250)} = 203,349$ req/s (Figure 15). Therefore, the minimum number of nodes to deploy is:

$$N_C = \left\lceil \frac{N_A T_A \overline{R}}{T_C^{(P)}} \right\rceil = \left\lceil \frac{50 * 40 * 4}{203,349} \right\rceil = 1$$

B) Scaling the system up to $10,000$ application nodes. Under the same conditions, if we scale the application tier up to 10k nodes ($N_A = 10,000$), we obtain a *Memcached* concurrency level $P = N_A \cdot p = 50,000$ connections. Considering that a single TCP connection requires up to 1 KB of memory in a Linux system, the memory overhead will be less than 50 MB per node. The performance of each *Memcached* server drops to $T_C^{(50K)} = 130,579$ req/s (Figure 15), due to resource contention caused by the high volume of requests. Thus, the number of *Memcached* nodes that are required to handle $10,000$ application nodes is:

$$N_C = \left\lceil \frac{N_A T_A \overline{R}}{T_C^{(P)}} \right\rceil = \left\lceil \frac{10,000 * 40 * 4}{130,579} \right\rceil = 13$$

We remark that this is a *worst-case* result. On average, a large part of the $50,000$ connections are idle most of the time: the concurrency level is lower than the number of active connections, since connection pools have spare connections.

Figure 16 shows the required number of nodes for an increasing number of application nodes (N_A) and for different values of $\overline{R}$, assuming that each application node has an engineered throughput equal to $T_A = 40$ req/s, as in the previous examples. When the number of application nodes is below 200, a single Memcached node suffices. In the extreme case of $10,000$ application nodes, in which a single application request performs 20 accesses on average, the solution requires up to 62 Memcached nodes, which is still less than 1% of the nodes of the application tier.

7. Related Work

In this section, we review previous studies on overload control. Since this topic is closely related to autoscaling and load balancing, we also review

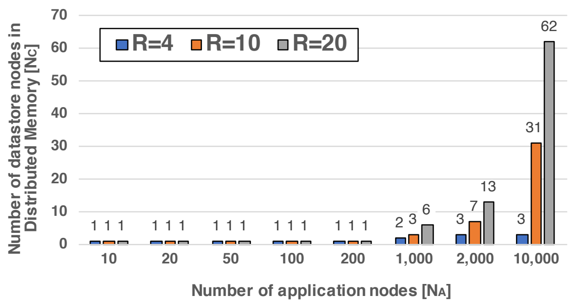


Figure 16: Number of nodes for the *Distributed Memory* (N_C), at different scales of the application tier (N_A) and average number of datastore operations per user request ($\overline{R}$).

studies in these areas and discuss how they are related to this work.

Autoscaling. A common strategy to face overloads is scaling-out cloud instances, either in a reactive way (e.g., when resources are exhausted, or delays are too long), or in a proactive way (e.g., by anticipating overloads through forecasting). Examples of reactive solutions are autoscaling solutions from public IaaS providers, such as AWS Auto Scaling [52], the UNIFY framework for NFV orchestration [53], which performs load balancing and elastic scaling of VNFs based on performance measurements, *Autoscale* [54], which scales out application nodes in response to workload pattern changes, and *VNF-DOC* [55], which auto-scales based on workload forecasting. However, these solutions are aimed at stateless cloud instances, but are not suitable for scaling stateful datastores, which need to redistribute data across nodes. Recent research on overloaded datastores includes *PAX* [56], which profiles the workload to detect and to redistribute hot-spot resources on newly added nodes. These solutions are complementary to our work for two reasons. First, scaling-out takes a non-negligible amount of time, during which the system is exposed to degraded QoS and software failures. In this timeframe, our throttling solution can cooperate with autoscaling, by preserving QoS of sessions up to the system capacity, while more capacity is added in background and data is redistributed. Second, overloads are caused not only by external workload surges, but can also be a consequence of internal faults, such as software bugs and misconfigurations. In these cases, scaling out is ineffective against the root cause of the overload.

Load balancing. Optimizing load balancing in datastores is another approach to prevent unbalanced overload conditions, by dynamically replicating and migrating data. *SPORE* [21] and *SP-cache* [24] address hot-spots due to skewed workloads, by taking into account key popularity to perform more sophisticated data replication. Zhang et al. [22] designed a middleware between applications and datastores, that includes a hot-spot detector, and a key redirector that replicates popular keys on multiple servers, and forwards them to servers according to their resources. *NetKV* [57] is a proxy that inspects key requests, and replicates hot-spot keys on multiple servers to mitigate unbalanced workloads. *MBal* [23] is an in-memory datastore designed to prevent load unbalancing problems. It includes a centralized coordinator that monitors the system state and performs data replication and migration. Similar to autoscaling, these strategies can be slow at adapting to sudden workload changes and faults, since data need to be copied across nodes. In contrast, throttling does not modify the way data are accessed and distributed, prevents unbalanced traffic in excess from entering into the system.

Throttling. Admission control and traffic throttling solutions have been frequently used in IT and telecom systems to promptly react to overloads. In general, these approaches monitor service performance (e.g., in terms of throughput and latency at the application layer) and resources (e.g., CPU utilization), and throttle traffic in a dynamic feedback loop. Kasera et al. [9] analyzed throttling algorithms in the context of carrier-grade telecom switches, such as the *Random Early Discard* (RED), which throttles traffic according to the request queue size, and the *Occupancy* algorithm, which throttles traffic according to CPU utilization and rate of accepted calls. Hong et al. [10] present a broad overview of overload control schemes for the SIP protocol. We adopted a similar algorithm for *NFV-Throttle* in the context of virtual network functions [25].

In the context of distributed systems, Welsh and Culler [32] proposed an adaptive overload control approach using a *token bucket* and a closed control loop to tune traffic according to service latency. This approach is quite representative of the state-of-the-practice in distributed applications, as in the case of the Clearwater IMS [40]. More recent developments for cloud computing include: *brownout* techniques [58, 34], which adaptively activate or deactivate optional parts of applications to manage overloaded resources; *DAGOR* [33], which performs business-oriented admission control by propagating a business priority level across requests for the same session, and tuning the admission level based on the current load; *Wisp* [35], which propagates local admission rates of individual processes, from downstream services to upstream ones, in order to perform admission control of entire workflows; and *Breakwater* [59], which reduces the overhead of overload control for microsecond-scale RPCs by issuing credits from downstream services to upstream ones, in a speculative way.

It is important to note that all of these studies addressed different aspects of overload control, but **none of them is applicable for handling data dependencies and hot-spots** that arise in multi-tier systems with large-scale datastores. Only few studies considered multi-tier web sites, such as Liu et al. [60], which adopt queuing theory and feedback loops for adaptive load control; and Muppala and Zhou [61] and Cherkasova and Phaal [62], which differentiate between user workload patterns to identify which tier can be overloaded. However, these solutions handle the tiers as a whole, and are not suitable against unbalanced overloads in large-scale datastores (i.e., affecting few specific nodes). Our solution mitigates this issue by performing throttling at a fine-grain, according to data dependencies and to the capacity of individual datastore nodes.

8. Conclusion

We analyzed unbalanced overload conditions in modern multi-tier systems with large-scale datastores, which can be caused by hot-spot resources, misconfigurations, background tasks, and hogs. We proposed DRACO, a distributed overload control solution that performs fine-grained requests throttling, by taking into account data dependencies and the available capacity of datastore nodes. This approach is suitable for multi-tier systems where traffic cannot be throttled by the inner tiers (e.g., due to data consistency issues); does not introduce changes in the datastore tier (e.g., it is applicable with off-the-shelf datastores); and it is based on simple, robust heuristics based on CPU and network utilization to tune the amount of traffic. We evaluated DRACO on two case studies: a distributed fileserver, which is sensitive to problems of data consistency and hot-spots, and a virtualized IP Multimedia Subsystem, which must withstand huge workloads and achieve high reliability. In our experiments, we simulated unbalanced overloads up to 100 times the capacity of the system. Results showed a significant improvement in both QoS and resource utilization.

Acknowledgments

This work has been partially supported by Huawei Technologies Co., Ltd., and by UniNA and Compagnia di San Paolo in the frame of Programma STAR.

References

[1] ETSI, Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action (2012).

[2] O. Ali, A. Shrestha, J. Soar, S. F. Wamba, Cloud computing-enabled healthcare opportunities, issues, and applications: A systematic review, International Journal of Information Management 43 (2018) 146–158.

[3] P. Somasekaram, R. Calinescu, R. Buyya, High-availability clusters: A taxonomy, survey, and future directions, Journal of Systems and Software 187 (2022) 111208.

[4] E. Bauer, R. Adams, Reliability and Availability of Cloud Computing, 1st Edition, Wiley-IEEE Press, 2012.

[5] G. Galante, L. C. de Bona, A survey on cloud computing elasticity, in: UCC, 2012.

[6] P. C. Brebner, Is your cloud elastic enough?: Performance modelling the elasticity of infrastructure as a service (IaaS) cloud applications, in: ICPE, 2012.

[7] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, I. M. Llorente, Scheduling strategies for optimal service deployment across multiple clouds, Future Generation Computer Systems 29 (6) (2013) 1431–1441.

[8] S. Sotiriadis, N. Bessis, P. Kuonen, N. Antonopoulos, The Inter-Cloud Meta-Scheduling (ICMS) Framework, in: Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on, 2013, pp. 64–73.

[9] S. Kasera, J. Pinheiro, C. Loader, M. Karaul, A. Hari, T. LaPorta, Fast and robust signaling overload control, in: ICNP, 2001.

[10] Y. Hong, C. Huang, J. Yan, A comparative study of SIP overload control algorithms, Net. Traff. Eng. Distr. App.

[11] A. Cockcroft, Netflix Global Cloud Architecture (2012). URL `https://www.slideshare.net/adrianco/netflix-global-cloud`

[12] A. Davoudian, L. Chen, M. Liu, A survey on NoSQL stores, ACM CSUR.

[13] L. Chi, X. Zhu, Hashing techniques: A survey and taxonomy, ACM CSUR.

[14] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup protocol for internet applications, IEEE/ACM TON.

[15] Y. Izrailevsky, NoSQL at Netflix (2011). URL `https://medium.com/netflix-techblog/nosql-at-netflix-e937b660b4c`

[16] M. Do, P. Oberai, M. Daxini, C. Kalantzis, Introducing Dynomite: Making Non-Distributed Databases, Distributed (2014).
URL `https://medium.com/netflix-techblog/introducing-dynomite-making-non-distributed-databases\-distributed-c7bce3d89404`

[17] S. Madappa, V. Nguyen, S. Mansfield, S. Enugula, A. Pratt, F. Siddiqi, Caching for a Global Netflix (2016).
URL `https://medium.com/netflix-techblog/caching-for-a-global-netflix-7bcc457012f1`

[18] D. Cotroneo, R. Natella, S. Rosiello, Dependability evaluation of middleware technology for large-scale distributed caching, in: IEEE ISSRE, 2020.

[19] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, R. Yao, Gray failure: The achilles' heel of cloud-scale systems, in: HotOS, 2017.

[20] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, et al., Fail-slow at scale: Evidence of hardware performance faults in large production systems, ACM TOS 14 (3).

[21] Y.-J. Hong, M. Thottethodi, Understanding and mitigating the impact of load imbalance in the memory caching tier, in: SoCC, 2013.

[22] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, H. H. Huang, Load balancing of heterogeneous workloads in Memcached clusters, in: FCW, 2014.

[23] Y. Cheng, A. Gupta, A. R. Butt, An in-memory object caching framework with adaptive load balancing, in: EuroSys, 2015.

[24] Y. Yu, R. Huang, W. Wang, J. Zhang, K. B. Letaief, SP-cache: Load-balanced, redundancy-free cluster caching with selective partition, in: IEEE SC, 2018.

[25] D. Cotroneo, R. Natella, S. Rosiello, NFV-Throttle: An overload control framework for Network Function Virtualization, IEEE TNSM.

[26] Softengi Ltd., AI in telecom industry – legacy systems modernization (2020).
URL `https://softengi.com/blog/ai-in-telecom-industry-legacy-systems-modernization/`

[27] Z. Avidan, H. Otharsson, Accelerating the digital journey from legacy systems to modern microservices, CreateSpace Ind. Pub. Platform.

[28] U. U. Hafeez, M. Wajahat, A. Gandhi, Elmem: Towards an elastic memcached system, in: IEEE ICDCS, 2018.

[29] D. Cotroneo, L. De Simone, R. Natella, NFV-Bench: A dependability benchmark for Network Function Virtualization systems, IEEE TNSM.

[30] R. Nishtala, H. Fugal, S. Grimm, et al., Scaling Memcache at Facebook, in: USENIX NSDI, 2013.

[31] Amazon Web Services, Inc., Understanding DynamoDB Adaptive Capacity (2018).
URL `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-partition-key-design.html`

[32] M. Welsh, D. E. Culler, Adaptive Overload Control for Busy Internet Servers, in: USENIX SITS, 2003.

[33] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, J. Yang, Overload control for scaling WeChat microservices, in: ACM SoCC, 2018.

[34] M. Xu, R. Buyya, Brownout approach for adaptive management of resources and applications in cloud computing systems: A taxonomy and future directions, ACM CSUR 52 (1).

[35] L. Suresh, P. Bodik, I. Menache, M. Canini, F. Ciucu, Distributed resource management across process boundaries, in: ACM SoCC, 2017.

[36] A. Beloglazov, R. Buyya, Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints, IEEE TPDS.

[37] J. Li, A. C. König, V. Narasayya, S. Chaudhuri, Robust estimation of resource consumption for SQL queries using statistical techniques, VLDB.

[38] Quality Excellence for Suppliers of Telecommunications Forum (QuEST Forum), TL 9000 Quality Management System Measurements Handbook 4.5, Tech. rep. (2010).

[39] ETSI, Network Functions Virtualisation: Infrastructure Overview (2015).

[40] Metaswitch Networks Ltd., Project Clearwater (2018).
URL `http://www.projectclearwater.org/`

[41] Y. Xu, E. Frachtenberg, S. Jiang, M. Paleczny, Characterizing Facebook's Memcached workload, IEEE Internet Computing 18 (2) (2013) 41–49.

[42] M.-C. Lee, F.-Y. Leu, Y.-P. Chen, Cache replacement algorithms for YouTube, in: IEEE AINA, 2014.

[43] J. Carroll, Building Pinterest in the cloud (2013).
URL `https://medium.com/@Pinterest_Engineering/building-pinterest-in-the-cloud-6c7280dcc196`

[44] J. Yang, Y. Yue, K. Rashmi, A large scale analysis of hundreds of in-memory cache clusters at Twitter, in: USENIX OSDI, 2020.

[45] I. Papapanagiotou, V. Chella, NDBench: Benchmarking microservices at scale, arXiv preprint arXiv:1807.10792.

[46] Metaswitch Networks Ltd., Clearwater performance and our load monitor (2015).
URL `http://www.projectclearwater.org/clearwater-performance-and-our-load-monitor/`

[47] Metaswitch Networks Ltd., Tuning overload control for telco-grade performance (2015).
URL `http://www.projectclearwater.org/overload-control-2/`

[48] R. Gayraud, O. Jaques, et al., SIPp: SIP load generator (2010).
URL `http://sipp.sourceforge.net/`

[49] Metaswitch Networks Ltd., Clearwater SIP Stress scenario (2016).
URL `https://github.com/Metaswitch/sprout/blob/dev/clearwater-sip-stress.root/usr/share/clearwater/sip-stress/sip-stress.xml`

[50] Amazon Web Services, Inc., AWS Cloud Storage (2021).
URL `https://aws.amazon.com/what-is-cloud-storage/`

[51] Amazon Web Services, Inc., Google Cloud Storage (2021).
URL `https://cloud.google.com/storage`

[52] Amazon Web Services, Inc., Introducing AWS Auto Scaling (2018).
URL `https://aws.amazon.com/about-aws/whats-new/2018/01/introducing-aws-auto-scaling/`

[53] R. Szabo, M. Kind, F.-J. Westphal, H. Woesner, D. Jocha, A. Csaszar, Elastic network functions: opportunities and challenges, IEEE Network 29 (3) (2015) 15–21.

[54] A. Gandhi, M. Harchol-Balter, R. Raghunathan, M. A.

Kozuch, Autoscale: Dynamic, robust capacity management for multi-tier data centers, ACM TOCS.

[55] S. Murugasen, S. Raman, K. Veezhinathan, VNF-DOC: A dynamic overload controller for virtualized network functions in cloud, in: AINA, 2020.

[56] S. Dipietro, G. Casale, PAX: Partition-aware autoscaling for the Cassandra NoSQL database, in: NOMS, 2018.

[57] W. Zhang, T. Wood, J. Hwang, NetKV: Scalable, self-managing, load balancing as a network function, in: ICAC, 2016.

[58] L. Tomás, C. Klein, J. Tordsson, F. Hernández-Rodríguez, The straw that broke the camel's back: Safe cloud overbooking with application brownout, in: IEEE ICCAC, 2014.

[59] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, A. Belay, Overload control for μs-scale RPCs with Breakwater, in: USENIX OSDI, 2020.

[60] X. Liu, J. Heo, L. Sha, X. Zhu, Adaptive control of multi-tiered web applications using queueing predictor, in: NOMS, 2006.

[61] S. Muppala, X. Zhou, Coordinated session-based admission control with statistical learning for multi-tier internet applications, JNCA.

[62] L. Cherkasova, P. Phaal, Session-based admission control: A mechanism for peak load management of commercial web sites, IEEE TC.