A control theoretical approach to non-intrusive geo-replication for cloud services

Dürango, Jonas; Tärneberg, William; Tomas, Luis; Tordsson, Johan; Kihl, Maria; Maggio, Martina

Published in:
2016 IEEE 55th Conference on Decision and Control, CDC 2016

DOI:
10.1109/CDC.2016.7798502

2016

A control theoretical approach to non-intrusive geo-replication for cloud services

Jonas Dürango[1], William Tärneberg[2], Luis Tomás[3], Johan Tordsson[3], Maria Kihl[2] and Martina Maggio[1]

[1]Department of Automatic Control, Lund University, Sweden

[2]Department of Electrical and Information Technology, Lund University, Sweden

[3]Department of Computing Science, Umeå University, Sweden

Abstract— **Complete data center failures may occur due to disastrous events such as earthquakes or fires. To attain robustness against such failures and reduce the probability of data loss, data must be replicated in another data center sufficiently geographically separated from the original data center. Implementing geo-replication is expensive as every data update operation in the original data center must be replicated in the backup. Running the application and the replication service in parallel is cost effective but creates a trade-off between potential replication consistency and data loss and reduced application performance due to network resource contention. We model this trade-off and provide a control-theoretical solution based on Model Predictive Control to dynamically allocate network bandwidth to accommodate the objectives of both replication and application data streams. We evaluate our control solution through simulations emulating the individual services, their traffic flows, and the shared network resource. The MPC solution is able to maintain a consistent performance over periods of persistent overload, and is quickly able to indiscriminately recover once the system return to a stable state. Additionally, the MPC balances the two objectives of consistency and performance according to the proportions specified in the objective function.**

I. Introduction

Today, there is an ever increasing reliance on cloud services for business critical operations. Outsourcing operational applications to one vendor expose businesses to potential revenue losses incurred by for example downtime due to failures [10]. In cloud computing, failures are the norm rather than an exception. Failures are unpredictable and may happen at any time, as exemplified by the cascading power blackout that swept cities from Detroit to New York City in 2003 [1]. The interruption in business continuity and the information lost when storage devices or a complete Data Center (DC) hosting an application fail can even put entire enterprises out of business [9]. Two out of five enterprises that experience a disaster are out of business within five years [11] from the outage. Furthermore, cost estimations for data unavailability can reach millions of Euros per hour [8]. These events and revelations have incited the development of Disaster Recovery (DR) schemes that provide reduced interruption of service in case of disasters.

Current DR schemes typically achieve redundancy by mirroring all relevant data on the application's primary operational node to one or multiple secondary replicas. The replicas are persistently standing by to assume the responsibility for hosting the applications, in the event the primary fails. In order to be tolerant to disasters severe enough to bring down an entire DC, such as a fire or an earthquake, replicas are kept geographically separated, known as geo-replication. We therefore consider a setup in which there are *two geographically distinct sites*. In order to achieve data replication, backups should be shared between the primary and the secondary site. As a result, the applications can stay available even as the primary replica is lost or becomes unreachable [8]. However, such DR solutions increase the overall network traffic from the primary node shared between the replication service and the applications. The additional traffic can lead to network contention between the tenants on the primary node during high loads. To mitigate this potential contention, the system administrators typically assign a static quota for the network bandwidth allotted to the replication service traffic. As an example, the Distributed Replicated Block Device (DRBD) replication tool documentation recommends a 30% bandwidth allotted to the replication service traffic[1].

Such solutions are inherently inflexible, as they do not cope well with irregular traffic patterns and the heterogeneous objectives of the different streams, manifested in their different goals. The replication service seeks to maintain the replicas as closely synchronized as possible to minimize potential data loss and unavailability in case of a failure. It does so by attempting to minimize the delay imposed to each write operation. On the other hand, the application traffic needs to be served at a certain rate to meet performance objectives, e.g., end-user response time. Therefore, there is an inherent *trade-off* between data consistency and delivered application performance, which strongly depends on the available bandwidth. We argue in this paper that these conflicting goals can be managed using a dynamic bandwidth allocation approach.

To this end, we propose a dynamic bandwidth allocation

Corresponding author: jonas.durango@control.lth.se. The research leading to the results presented in this paper has received funding from the European Union's seventh framework programme (FP7 2007-2013) Project ORBIT under grant agreement number 609828. In addition, this work was partially supported by the Swedish Research Council (VR) for the projects "Cloud Control" and "Power and temperature control for large-scale computing infrastructures". J. Dürango, W. Tärneberg, M. Maggio, and M. Kihl are members of the LCCC Linneus Center and ELLIIT Excellence Center at Lund University. W. Tärneberg is also funded by the Mobile and Pervasive Computing Institute Lund University (MAPCI). The work was done while W. Tärneberg was hosted by the University of Virginia.

[1] https://drbd.linbit.com/en/users-guide/s-configure-sync-rate.html

solution for DR systems based on Model Predictive Control (MPC). We propose a two fold solution: (1) differentiating between different traffic flows and concurrently use different replication modes; and (2) an MPC solution that dynamically adjusts resource allocations for the different traffic flows over time based on the prevailing conditions in an attempt to meet their individual performance objectives. Our proposed solution dynamically adapts to changes in egress (outgoing) traffic and provides an optimized scheduling of bandwidth, in order for the replication service traffic to be handled with low latency at the same time as the ordinary application traffic can keep its performance objectives. The fundamental principal is a holistic one, we allow the controller to compromise the throughput of one of the services while maintaining its performance goals to meet the objectives of achieving better overall system performance and cost-efficiency, with the resources at hand. We validate our strategy with a simulator that executes a variety of workloads and measures the amount of data loss in case of a disaster. Finally, to evaluate the replication performance of our solution, we formulate a performance metric that captures the momentary disaster recovery readiness of the system.

II. Related work

In this section we provide an overview of fault tolerance and disaster tolerance techniques for cloud services, and of replication challenges in general. We begin by discussing prevalent replication techniques, their inherit challenges, and the current state of the research in that area. We then tie it into the problem we are addressing.

Making DR cost-efficient is a significant research area [8]. More and more companies are focusing on recovery plans, in an attempt to achieve what is generally known as business continuity [13]. The main principal of business continuity is to offer application owners the assurance that their applications will have as few service interruptions as possible. To achieve this, many business services utilize (1) fault tolerance techniques and (2) disaster recovery techniques. Fault tolerance techniques, such as Remus [3] or COLO [5] are used to recover from sporadic failures by synchronizing what a VM is doing into a secondary copy of the VM.

Other work highlight the importance of data replication and try to reduce the incurred cost of replication. One example is [2]. In this case, the main objective is data durability and how to protect against both independent and correlated node failures by means of a tiered replication scheme that splits the cluster into a primary and a backup tier. Regarding disaster recovery techniques, in [13] the authors propose to use a public cloud to recover in case of a disaster instead of a backup site.

Replication incurs additional overhead during the normal execution of the DC. In general, in response to client-issued requests, applications continuously write data onto their attached virtual disks. As part of the DR solution, a replication service is then responsible for mirroring the write operations at the secondary replica. Such mirroring can be carried out by either synchronous or asynchronous write operations. Synchronous writes provide a higher degree of data consistency between replicas as each write operation at the primary replica has to be verified to have been carried out also at the secondary replica before completing. Pipecloud [14] is a synchronous backup strategy that addresses the impact of replication latency on performance by efficiently overlapping replication with application processing for multi-tier servers. However, as write operations must await response from the backup site before completing, synchronous backup guarantees consistency at the expense of collocated services sharing the same resource. The primary replica is essentially allowed to pull ahead of the secondary replica by completing write operations when they have been made to the local file system, without waiting for the secondary replica. The replication service is then responsible for carrying out the write operations at the secondary replica.

This is especially benefician in a Metropolitan Area Network (MAN) or Wide Area Network (WAN) setting, where bandwidth limitations and high latency can make replication unacceptably slow, as the network connectivity between replicas becomes a performance bottleneck. Performance are then improved at the expense of consistency guarantees by using asynchronous replication. This clearly creates some inconsistencies, until the write operations have been propagated to the secondary replica, but at the same time avoids performance bottlenecks. For instance, in SnapMirror [10], batches of updates are periodically sent to the backup site, aiming at trading off cost and performance. SnapMirror's asynchronous solution does, however, not offer continuous mirroring but only guarantees that the copies are in sync when backups are performed. The degree of replica consistency is thus proportional to the delay incurred by the intermediate network and the availability of shared resources.

One frequently employed service for replicating file systems in Linux systems is DRBD [12] and DRBD Proxy, which have support for both synchronous and asynchronous replication modes. The DRBD [12] asynchronous replication mode sends data continuously but only waiting for the acknowledgement that the packages has reached the send-TCP buffer in the local server, unlike synchronous mode that waits for the acknowledgement of the write operation at the remote location. DRBD is our choice as an enabling technology for the design of our DR solution.

As regards to the interference between the replication service traffic and the normal DC operation, besides the well known techniques to differentiate traffic flows at routing level (such as DiffServ or IntServ[2]), there are tools available for traffic sharing, allowing traffic differentiation per process or flow at server level. For instance, Dusia et al. present a network quality of service guaranteeing approach [6] capable of prioritizing some processes (in their case containers) by making use of the Linux traffic control (TC) utility. However, they define an static setting, not aware of current buffer status or data flow needs.

[2] https://tools.ietf.org/html/draft-ietf-diffserv-rsvp-02

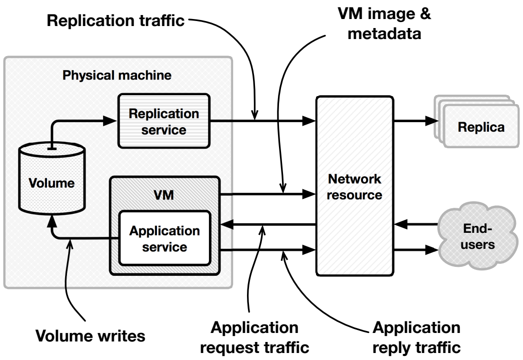

Fig. 1. System abstraction for the set-up considered from the primary replica's viewpoint. The application replies to incoming client requests, while the replication service is responsible for mirroring the data written by the application as well as the state of the VM at the remote replica. The application and replication service share a common network resource.

III. System architecture model

In this section we outline the system architecture model used in this paper. Emphasis is given to describing traffic streams and system components through which they can be managed. We consider applications that are hosted in a primary replica executing in either a Virtual Machine (VM) or container that in turn is hosted on a Physical Machine (PM) in a DC. Requests from clients are received by the application, in turn prompting the computation of responses that are returned to the issuers.

A. *Dynamic Control for Concurrent Flows Transmission*

Apart from mirroring the data that the application writes during runtime, the DR service also needs to transfer the information regarding the VMs running the application, including the VMs image and meta-data on its current state such as virtual disks attached (known as volumes) and network configurations. As such configurations usually are not frequently updated, the transfer of the corresponding data to the remote site is usually initiated at set time intervals. Figure 1 provides an overview of the system.

To reduce cost, the replication service is not given a dedicated interface for its traffic. Instead, the network resources are shared between the application and replication service. This introduces inherent conflicting goals, since the Quality of Service (QoS) of an application is typically directly related to the rate at which it can serve requests from clients. On the other hand, the degree to which the application remains disaster tolerant is subject to the rate at which the application write operations can be mirrored to the remote replica and how expediently the transfer of VM images and related meta-data can be completed. By not considering this trade-off, high load situations can lead to unacceptable service degradation or disaster tolerance. On the other hand, existing solutions to addressing said trade-off can prove too inflexible in a dynamic setting where traffic conditions are subject to unpredictable changes. Hence the need for a dynamic solution.

B. *Flows Differentiation and Traffic Model*

The solution proposed in this paper builds on differentiating the three traffic flows described previously, denoted *Application*, *Replication service*, and *VM image*, and taking into account their different time-variant features when managing them. The approach considered here is black-box, agnostic to and decoupled from the actual application and the replication service. This is to make the approaches as general and as portable as possible, and to facilitate easier deployment in a future test-bed. Separating the solution from the nature of the application means that traffic sent by the *Application* and *Replication service* from the traffic management component's point of view can be seen as exogenously generated. The *Application* and *Replication service* traffic streams generally have time-varying rates at which data need to be sent. In this paper, we assume the traffic streams to be non-homogeneous Poisson processes [7]. The *VM image* flow is also exogenous to the traffic management, but for reasons outlined earlier it can be assumed to instead arrive in bulks at given time intervals. An illustration of the structure of the traffic management solutions discussed here is given in Figure 2.

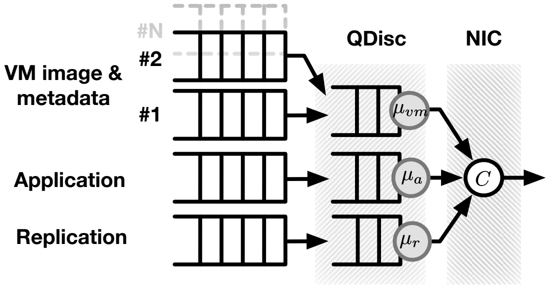

Fig. 2. Structural overview of system components relevant to traffic management. The *Replication service*, *Application* and individual *VM image* copy streams are differentiated. Multiple VM image copies are kept differentiated from each other. Streams try to deposit packets in target QDisc buffers and wait if the target buffer is full.

When there is data ready for transmission, the system needs to decide which differentiated traffic stream gets access to the network resource. In Linux and other operating systems, egress network traffic can be managed, policed, and shaped through a QDisc [6], which acts as a scheduler on the outgoing interface. By default most systems perform no traffic differentiation and the QDisc acts as a simple FIFO buffer. This can be disadvantageous in some circumstances as it allows one traffic stream to grab an unproportionally large share of the bandwidth by sending packets at a high rate. A more flexible alternative, QDiscs such as Hierarchical Token Bucket (HTB) [4] employ filters to give the system administrator large freedom in managing outgoing traffic per traffic type. Combining differentiation to classify traffic streams and HTB filters, individual traffic streams can be allotted a share of the network bandwidth over a certain time period, and bandwidth sharing hierarchies can be constructed. This way, each traffic stream is guaranteed to receive its share of the bandwidth and is not vulnerable to bandwidth hoarding streams in the way the FIFO solution is. Yet another commonly used approach for traffic management is to prioritize traffic streams, where higher ranked streams persistently pre-empts lower ranked streams. This is supported by default in the QDisc structure in most Linux distributions, `pfifo_fast`, which combines FIFO scheduling with three priority levels. Traffic streams are given a priority, and packets are buffered in three FIFO buffers, one for each priority level. The QDisc then schedules packets for transmission from buffers in falling priority order. This type of traffic management is ideal for allowing interactive latency-sensitive applications transmitting relatively little data to still access the network resource while larger bulk-type transfers also are active.

It is worth noting that by their nature, buffer sizes of QDiscs are relatively small, typically ranging from kilobytes to a few megabytes. Accordingly, the QDisc buffers cannot be expected to be able to accommodate all traffic at all times, in particular during sudden bursts and when the system is intermittently overloaded. In a real system, when a buffer fills up, the transportation layer would incur *back pressure* in the QDisc, thereby forming a closed loop system where the rate at which packets are deposited into the buffer is reduced to match the rate at which it is emptied. In this paper we do not explicitly consider back pressure as it goes against the application agnostic approach taken. Instead each traffic stream is kept in its own infinite buffer that deposits its content into the target QDisc buffer. For *VM image* traffic this has some implications. In our set-up, whenever the DR system schedules a VM image copy, it is treated as a bulk arrival of packets that are deposited in its own buffer, which in turn tries to make deposits in the target QDisc buffer. If a copy is scheduled to start before a previous one is finished, two streams would try to deposit in the same target buffer. Effectively, this halves the potential bandwidth available to each VM image copy stream until one of them is finished. It is again worth noting that as the proposed method is agnostic to the hosted services it cannot control the arrival of VM image copies.

IV. Control Strategy

This section describes our proposal for dynamically adjusting bandwidth shares to the traffic streams identified in Section III. It is based on the HTB filter approach, where traffic streams are differentiated and allocated a guaranteed minimum share of the available network bandwidth. Typically, the guaranteed minimum share is set statically depending on some knowledge of the system requirements. In contrast to that, our MPC controller dynamically adjusts the guaranteed minimum share for each of the traffic stream using feedback of the current state of the system.

Let $\lambda_i(k)$, $i \in I = \{a, r, vm\}$ denote the amount of data that the DC is requested to transmit in the sampling interval $[k, k+1]$ for each of the streams: *Application* (a), *Replication service* (r) and *VM image* traffic (vm). These requests are considered to be exogenous to the traffic management system. Also, based on the total available network bandwidth for the primary replica, let C denote the total amount of data that can be sent during a sampling period, C being dependent on the DC network link.

We denote with u_i the control signal that we use, which is the fraction of network bandwidth reserved for each of the traffic streams. We actuate that via a minimum share of the bandwidth $Cu_i(k)$, $u_i = \{u_i | u_i \geq 0, \sum u_i = 1\}$. It is important to notice that u_i is only a minimum guaranteed share, which helps us avoiding wasting available bandwidth[3]. Bandwidth left unused by streams that did not have enough data to transmit can then be used by other streams, thus maximizing total bandwidth utilization. Conversely, if the allocated bandwidth for a stream is insufficient to complete the transmission, the exceeding data is buffered.

Let $x_i(k)$ denote the buffer levels at time k for each traffic stream, i.e., the data that is ready to be sent at time k for each stream. For each of the streams, $\forall i \in I$, we can then define the following linear integrator dynamics for the system:

$$x_i(k+1) = x_i(k) + \lambda_i(k) - Cu_i(k) - d_i(k). \tag{1}$$

In Equation (1), the disturbance terms $d_i(k)$ model actions that are not in direct relationship with the control signal, for example taking into account the situation in which buffers are emptied because there was not enough traffic in one of the other streams. The *actual sent traffic* for traffic stream i in the time interval $[k, k+1]$ is therefore:

$$\mu_i(k) = Cu_i(k) + d_i(k). \tag{2}$$

We assume the buffer levels to be measurable[4]. Indeed, in real implementations, a measurement of the amount of data sent per traffic stream $\mu_i(k)$ is usually also available. Using that, measurements of the data arrival processes $\lambda_i(k)$ can be reconstructed as follows:

$$\lambda_i(k) = x_i(k+1) - x_i(k) + \mu_i(k). \tag{3}$$

We model the arrival processes as standard input disturbances. VM image traffic is modeled as impulses arriving at fixed intervals while for Application and Replication service traffic one of two possible disturbance models is used. In one case, traffic is assumed to be slowly varying, with the following state space representation:

$$\begin{aligned} z_i(k+1) &= z_i(k) + e_i(k) = Fz_i(k) + e_i(k) \\ \lambda_i(k) &= z_i(k) + v_i(k) = Gz_i(k) + v_i(k). \end{aligned} \tag{4}$$

In the second case, we extend the previous model with a local linear trend, with corresponding state space representation:

$$\begin{aligned} z_i(k+1) &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} z_i(k) + e_i(k) = Fz_i(k) + e_i(k) \\ \lambda_i(k) &= \begin{pmatrix} 1 & 0 \end{pmatrix} z_i(k) + v_i(k) = Gz_i(k) + v_i(k), \end{aligned} \tag{5}$$

where $e_i(k) \sim N(0, \Sigma_e)$ and $v_i(k) \sim N(0, \Sigma_v)$. A Kalman filter is used to estimate the states of the disturbance models, which are then used as initial conditions for predicting future traffic by the MPC controller. Among the traffic streams, *VM image* stands out in the sense that the marginal benefit from allocating bandwidth to it is zero up until the point the transfer of a full image is completed. Bandwidth used to serve an image transfer without completing it is therefore

[3] In other words, the traffic shaping is *work preserving*, meaning that shares are only enforced if there is enough data to transmit for each traffic stream.

[4] The lack of distinction between traffic stream buffers and QDisc buffers in Equation (1) is due to the fact that traffic streams are differentiated. This means that they are the only actor depositing packets in their target QDisc buffers. Therefore it is possible to aggregate data residing in each QDisc buffer and model it as one larger, measurable, buffer.

essentially wasted. For this reason, we augment the system description with an integral state i_{vm} for the VM image buffer to incentivize the controller to finish image transfers. By setting $F_d = \mathrm{diag}(F,\ F)$ we get a complete state space description of the system, augmented with corresponding disturbance and integral states, as

$$x(k+1) = \begin{pmatrix} I & G_d & 0 \\ 0 & F_d & 0 \\ Z & 0 & 1 \end{pmatrix} x(k) - C \cdot \begin{pmatrix} I \\ 0 \\ Z \end{pmatrix} u(k) - \begin{pmatrix} I \\ 0 \\ Z \end{pmatrix} d(k), \tag{6}$$

$$G_d = \begin{pmatrix} G^T & 0 & 0 \\ 0 & G^T & 0 \end{pmatrix}^T, Z = \begin{pmatrix} 0 & 0 & -1 \end{pmatrix},$$

with the state vector $x = \begin{pmatrix} x_a & x_r & x_{vm} & z_a & z_r & i_{vm} \end{pmatrix}$. We then use Equation (6) in our MPC controller design to predict the evolution of the system, assuming that the disturbances d_i are zero-mean and uncorrelated.

In the design of the MPC controller, we use a standard quadratic cost function with penalties on buffer sizes and control signal variations $\Delta u_i(k) = u_i(k) - u_i(k-1)$,

$$J = \sum_{k=1}^{H_p} \sum_{i \in I} \left(q_i x_i^2(k+1) + r_i \Delta u_i^2(k+1) \right) + q_n i_{vm}(k+1). \tag{7}$$

H_p is here the prediction horizon, while q_i and r_i are the penalties on buffer lengths and control signal variations, respectively and q_n the penalty on the integrator state. Neither buffer lengths nor control signals can be negative, so those properties enter as natural constraints to the problem.

The controller formulation then takes the form

$$\begin{aligned} \text{minimize} \quad & \sum_{k=1}^{H_p} \sum_{i \in I} \left(q_i x_i^2(k+1) + r_i \Delta u_i^2(k+1) \right) \\ & + q_n i_{vm}(k+1), \\ \text{subject to} \quad & \text{Equation (6)}, \\ & 0 \leq x_i \leq \bar{x}_i, \\ & u_i \geq 0, \\ & \sum_{i \in I} u_i \leq 1, \end{aligned} \tag{8}$$

where the upper limits $\bar{x}_i$ on buffer levels represent a tunable maximal amount of buffered data we can tolerate for each traffic type. The controller then selects $\Delta u_i(k+1)$ and therefore $u_i(k+1)$ in order to minimize the cost function. In the cost function, we decided to minimize Δu as well, to avoid incurring into the effect of the dynamics of the TCP window and other unmodeled dynamics. The approach allows us to trade data consistency (bandwidth share given to the *Replication service* and the *VM image* traffic) and performance (bandwidth assigned to the *Application*).

V. Evaluation

This section discusses the evaluation of the proposed solution and the comparison of the results obtained with the MPC controller, and comparing them with the other alternatives we introduced in Section III. We tested the different strategies with a simulator and in many different scenarios, two of which are reported in the following. To compare the results, we have identified three metrics that summarize the behavior of the system and permit a comparison of the solutions. At the end of this section, we present some general conclusions that can be drawn from the experiments shown in this paper and from our experience with other scenarios.

A. Simulation Framework

In order to evaluate our proposed solution we have designed an event-based simulator using Python and SimPy[5]. The simulator is based on the system model detailed in Section III. It includes implementations of a set of alternative traffic management solutions, the foundations of which are outlined in Section III, together with the MPC controller introduced in section IV. The policies that complement our solution are the following:

- In *FIFO*, all traffic streams deposit packets in a shared QDisc buffer that is served by the network resource in a FIFO manner. When the buffer is full, the packet waits until further space in the buffer is available. This particular strategy mirrors a system's default behavior when no deliberate traffic shaping effort has been made by the system administrator.
- The *STATIC* solution implements a static bandwidth assignment, similar to the HTB filter approach described in Section III. Each traffic stream is guaranteed a set share of the network bandwidth at all times. In our case we devote 30% of the bandwidth to the *Replication service* traffic, following the guidelines for DRBD. For the *VM image* traffic, we calculate the amount of bandwidth necessary to finish a session copy before the next is initiated. The remaining bandwidth is devoted to the application traffic.
- The *PRIO* strategy relies on priorities, assigned to each traffic stream. The priorities are fixed and assigned by the system administrator, based on a ranking of which traffic stream would benefit most from receiving prioritized access to the network. In the simulator, we have given the highest priority to the *Application* traffic in order for it to be minimally negatively impacted by the presence of *Replication service* traffic. The second highest priority is given to the *Replication service* traffic, with *VM image* traffic having the lowest priority. As previously described, each priority level has its own FIFO buffer in the QDisc that is served by the network resource only if higher prioritized buffers are empty.

All these solutions are work preserving, thereby maximizing bandwidth utilization. The total traffic is therefore the same with all the solutions, the difference being how much of the shared resource is allocated to the different traffic types.

B. Performance metrics

In order to evaluate the behavior of each bandwidth allocation strategy, we perform simulations recording a set of relevant performance metrics. The set of metrics assesses the behavior of the traffic shaping solutions along different axes: the performance delivered to the application, the traffic needed for replication purposes, and the amount of data lost in case a disaster happens, it being data that has been buffered for replication but never sent out.

[5] http://simpy.readthedocs.org

For *Application* and *Replication* streams, we observe the mean level i_μ and 95th percentile $i\_\lambda^{0.95}$, $\forall i \in \{App, Rep\}$ of buffered traffic over the entire experiment. The *VM image* transfer process is evaluated based on the average vm_σ_μ and 95th percentile vm_σ_λ of the time passed since the creation of the most recent *VM image* available at the secondary replica. This reflects the state to which a system could roll back in case a disaster happens. We also observe the mean vm_μ and 95th percentile vm_λ of the transfer times for the completed *VM image* transfers.

We evaluate the application performance based on the **waiting time** spent in the system by each packet that belongs to the *Application* stream. To provide a measure of the effort required to restore a service following a disaster, at each point in time we take the last available *VM image* at the backup site and we sum the amount of write operations that have been made since the timestamp associated with that image. This gives us an indication of the amount of data that should be recreated in case a replica should be fired up at a third site. We refer to this metric as the **Disaster Recovery Overhead (DRO)**. Finally, we measure the amount of data currently in the replication buffer. This data is considered lost at the moment of a failure (**Data loss**) since it has not been transferred to the replication site.

C. Experiment 1

Scenario: In this first experiment, we explore the behavior of the system in a 3-hour long experiment. In this experiment, the traffic mix changes slowly, producing periods in which the DC is overloaded and periods in which the network capacity is enough to serve the incoming traffic and the replication traffic. More specifically, the significant contributor to overload alternates between *Application* traffic and *Replication service* traffic. This traffic composition models the normal operation of a DC with which daily patterns (for example, a news website usually receives more visits during the lunch break).

The total available bandwidth to the system C is 100 Mbps, and the *Application* outputs on average 62.5 Mbps, while *Replication service* operations are made on average at 32.5 Mbps. Every 10 minutes, a *VM image* copy is initiated with a fixed size of 375 MB, which corresponds to an average rate of 5 Mbps over a 10 minute period. The first plot in Figure 3 shows the *Application*, *Replication service* streams and the moments in which *VM images* are transmitted.

We configure STATIC and PRIO as outlined in Section V-A. For the MPC, we set the state penalties to $(q_a,\ q_r,\ q_{vm},\ q_n) = (600,\ 250,\ 50,\ 1)$ and for control signal variations $(r_a,\ r_r,\ r_{vm}) = (10^6,\ 10^6,\ 10^6)$ with the prediction horizon $H_p = 30$, corresponding to 5 minutes as the sampling time is 10 seconds. The slowly varying traffic arrival model outlined by Equation (4) is used, although we found during the experiments that the performance of the MPC controller was largely unaffected by the choice of traffic model. Lastly, FIFO is configuration-free.

Results: Figure 3 illustrates the metrics for Experiment 1, while Table I presents the resulting statistics of the experiment's outcome. The second plot in Figure 3 shows the waiting time for the read operations. At times, the DC is overloaded and there is not enough bandwidth to transmit the data belonging to all the streams. In this case, the time that each packet belonging to the *Application* stream waits in the system becomes larger. The FIFO strategy results in a significant increase of the waiting time, therefore reducing application performance. With FIFO, the system recovers only after a large enough period of under load is experienced. Conversely, when employing STATIC as traffic shaping mechanism the *Application*'s performance is quickly able to recover as soon as the arrival rate for the read traffic does not exceed the static capacity allotted to it. The PRIO is also penalized during overload — for example in the time interval $t = [1800, 4200]$. However, the *Application* penalty is only due to the overload generated by the *Application* traffic itself, which temporarily exceeds the capacity of the DC. The proportionality of the overload contributed by the *Application*

TABLE I
STATISTICS FOR EXPERIMENT 1

Application	MPC	FIFO	STATIC	PRIO
App_μ [MB]	1616	3077	2076	183
$App\_\lambda^{0.95}$ [MB]	6988	8185	8100	1245
Replication				
Rep_μ [MB]	3442	2336	3168	2976
$Rep\_\lambda^{0.95}$ [MB]	6989	7944	9863	6807
VM image				
vm_σ_μ [sec]	1203	377	607	5388
vm_σ_λ [sec]	2275	650	931	10126
vm_μ [sec]	935	81	321	5211
vm_λ [sec]	1881	90	433	9992

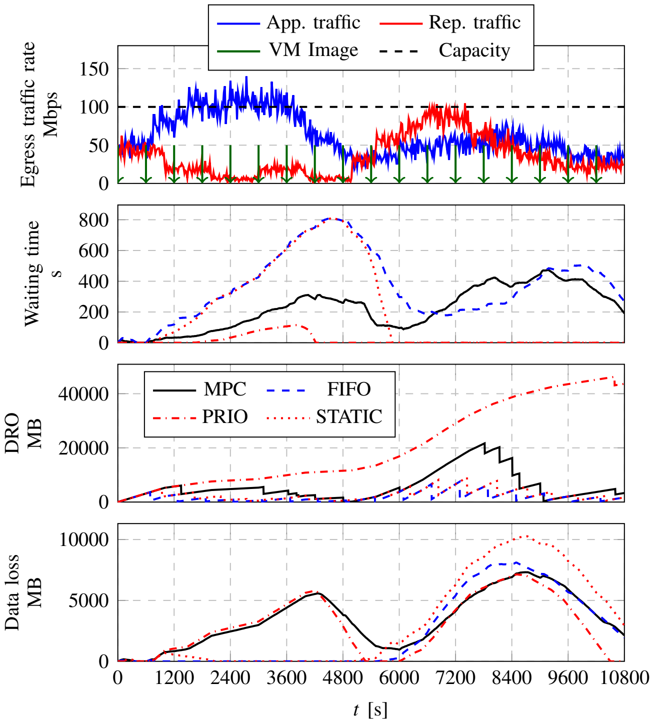

Fig. 3. Results from experiment 1. Top figure shows the rates at which *Application*, *Replication* and *VM image* traffic arrives. Next figure shows the recorded waiting time before being sent for *Application* traffic. Thereafter the DRO is shown, while the potential data loss in case of a disaster is shown at the bottom.

is a common factor for both the STATIC and PRIO methods, although the waiting time is penalized to different extents. The MPC solution here provides a middle-ground with acceptable buffering proportional to the aggregate overload. The MPC is able to indiscriminately accommodate both types of overload (read arrival rate exceeding the capacity and total arrival rate exceeding the capacity) with a consistent level of *Application* performance.

The third plot in Figure 3 shows the DRO. As can be seen, FIFO and STATIC are able to accommodate the replication traffic and provide good replication performance. On the contrary, the priority given to the read traffic for PRIO comes at a significant DRO. Not only does the overhead fast exceed any other method but it diverges in this time-frame. The MPC solution is able to quickly recover also in terms of DRO. The last plot of Figure 3 shows the amount of data that is not recoverable in case a disaster happens at a specific time. When the read traffic is generating the overload conditions, only the PRIO and MPC method suffer from the possibility of data loss. On the contrary, when the write traffic is higher than the static channels allocated for FIFO and STATIC, the data loss of all the alternatives are comparable. In Table I it is possibile to see that while PRIO is the best in terms of *Application* performance, the MPC controller is the second best (App_μ, $App\_\lambda^{0.95}$), with STATIC and FIFO not being a good fit to handle the read traffic. While FIFO is good on average for *Replication* performance (Rep_μ), it is not consistently better (the 95th percentile $Rep\_\lambda^{0.95}$ is higher than with PRIO and MPC). The MPC solution is better at exploiting the trade-off between different traffic conditions, and is able to trade consistency for performance and viceversa. FIFO and STATIC are the best at transfering the *VM images*, while PRIO is unable to handle this part of the traffic (vm_σ_μ, vm_σ_λ, vm_μ, vm_λ).

D. Experiment 2

Scenario: In this second scenario we show long periods of stable traffic levels inter-spreaded with abrupt changes with resulting high and low network load. This could for example correspond to an application switching between different operating modes — *e.g.*, computing statistics and applying changes to the data. In contrast to the previous experiment, the overload in this scenario is less extreme. Here, the contribution to the contention is more uniform across the traffic types. The simulation experiment is run for total duration of two hours, and the total available bandwidth to the system C is 100 Mbps. *Application* traffic arrives at an average rate of 71 Mbps, *Replication service* operations to be replicated at 22 Mbps, and *VM image* copies are again initiated every 10 minutes with an image size of 375 MB. The various policies are configured as for the previous experiment. The STATIC shares are equal to the traffic rates, as if the operator could perfectly know the traffic composition. For the MPC we use the penalties $(q_a,\, q_r,\, q_{vm},\, q_n) = (10^4,\, 2\cdot 10^3,\, 3\cdot 10^4,\, 1)$ and $(r_a,\, r_r,\, r_{vm}) = (5\cdot 10^6,\, 5\cdot 10^6,\, 5\cdot 10^6)$, while the prediction horizon is again $H_p = 30$. In this case we used the local linear traffic model described by Equation (5), but as in the previous case we found that this choice did not affect the performance of the MPC controller by much.

TABLE II
STATISTICS FOR EXPERIMENT 2

Application	MPC	FIFO	STATIC	PRIO
App_μ [MB]	32	386	306	0.33
$App\_\lambda^{0.95}$ [MB]	100	1089	1083	1.03
Replication				
Rep_μ [MB]	222	23	18	86
$Rep\_\lambda^{0.95}$ [MB]	681	223	84	401
VM image				
vm_σ_μ [sec]	939	369	642	1231
vm_σ_λ [sec]	1720	643	975	3019
vm_μ [sec]	680	75	374	1003
vm_λ [sec]	1460	79	429	2748

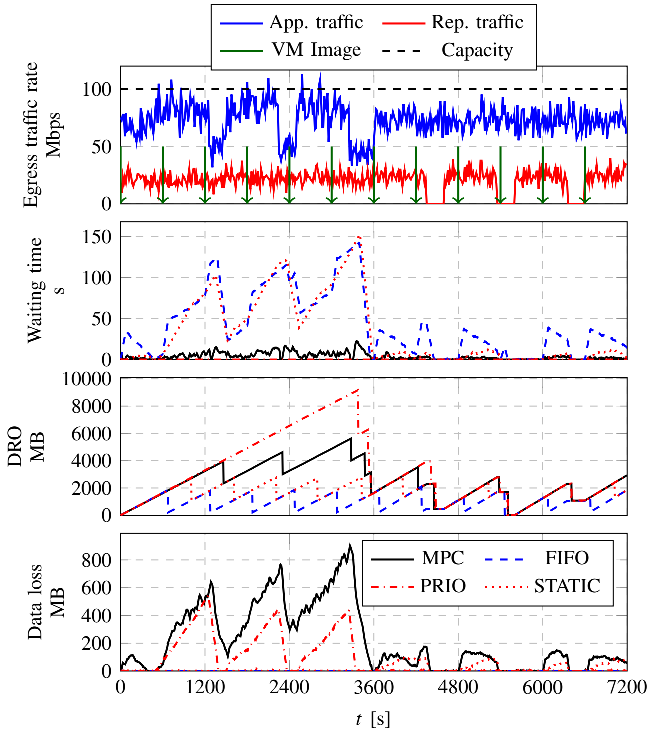

Fig. 4. Results from experiment 2. Figures as in Figure 3.

Results: The results from this experiment are summarised in Figure 4 and Table II. The second plot in Figure 4 shows the waiting time for the read requests, while the third plot shows the DRO. Both FIFO and STATIC sacrifice the *Application's* performance in favour of significant *Replication service* traffic. As a result, both of these methods persistently achieve the lowest disaster recovery overhead. In this scenario the *Application* traffic generally does not exceed the capacity C. As a result the PRIO method almost fully accommodates the *Application* across the entire observed time period. However, the PRIO method results in a very high DRO, which recovers only when the *Application* traffic is significantly below the system capacity, with significant lag.

In the observed scenario, the MPC method is able to maintain a negligible *Application* performance degradation under periods of overload. The MPC solution's ability to balance the two objectives is made clear by the small sacrifice in *Application* performance for a significant reduction of disaster recovery overhead. This specific ability to negligibly sacrifice *Application* performance also contributes to accelerating the recovery of the momentary disaster recovery overhead once the system return to an aggregate stable load, proportionally regardless to the composition of the load. Table II confirms these results.

E. Summary of findings

The primary objective of this evaluation is to determine the effectiveness of a dynamic solution to cope with the different type of traffic combinations and changes that happens in a real DC. The second aim of the experimental analysis is to determine how our MPC solution fits as a means to this end.

After trying a multitude of workloads, we can conclude that some of these workloads highlight features and weaknesses of all the different traffic scheduling solution that we have described. The FIFO solution has proven to be effective at accommodating all applications needs for streams when underloaded, and indiscriminately penalising when overloaded. In particular, in the scenarios we have rendered, a large portion of the traffic is *Application* traffic, which makes FIFO unsuitable since the end users suffer from buffering thus inhibiting the end-to-end performance of the *Application*. On the other hand, with FIFO, the replication traffic and the *VM image* traffic are able to indiscriminately gain access to the shared resource and are therefore served with a reasonable and fair delay which is in line with queuing theory findings [7]. Furthermore, the STATIC solution manages to isolate the *VM image* traffic and guarantee that the images are transferred timely, but is sensitive to any changes in the other traffic streams. Its evident inability to accommodate the individual objectives of the tenants make FIFO unsuitable for this system. The PRIO solution is inherently the most successful in terms of accommodating *Application* performance, but fails at accommodating the other tenant's objectives. In most of the scenarios we have run, it performs well for the *Replication service* traffic but fails at containing and recovering the momentary disaster recovery overhead in a timely manner.

From the experiments above, we can conclude that our dynamic method is the best to achieve a desirable balance between *Application* performance and disaster fault tolerance readiness in an intermittently overloaded system. Furthermore, the MPC method is able to capture the trade-off between delivering acceptable *Application* performance and accommodating the *Replication service*. The MPC solution is able to maintain the most consistent performance over periods with persistent overload, and is quickly able to indiscriminately recover once the system return to a stable state. Additionally, the MPC is able to persistently balance the two objectives according to the proportions specified in the objective function.

VI. Conclusion and Future Work

In this paper we design an MPC controller to determine the amount of bandwidth to be allocated to different streams in a cloud computing infrastructure. Our investigation starts from the detection of an inherent trade-off between data consistency in case of disasters and performance delivered by applications to end users.

In fact, the outgoing bandwidth in the data center is used concurrently both to replicate the changes operated by the users in the secondary backup, targeting consistency, and to respond to the user requests, targeting performance. The available outgoing bandwidth is however limited. So, while there is a desire to serve the user requests timely, it is also important to ensure that the amount of data loss in case of a disaster is limited.

We have developed a dynamic solution for this problem, in the form of an MPC controller, that we compared to the static solutions that are currently the best practice. The result of our investigation is that a dynamic solution is more flexible and it is capable of exploiting the mentioned trade-off.

References

[1] J. Barron. *The blackout of 2003: The Overview; Power surge blacks out northeast, hitting cities in 8 states and Canada; Midday shutdowns disrupt millions.* 2003. URL: http://www.nytimes.com/2003/08/15/nyregion/blackout-2003-overview-power-surge-blacks-northeast-hitting-cities-8-states.html.

[2] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. "Tiered Replication: A Cost-effective Alternative to Full Cluster Geo-replication". In: *2015 USENIX Annual Technical Conference (ATC 15)*. 2015, pp. 31–43.

[3] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. "Remus: High Availability via Asynchronous Virtual Machine Replication". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2008, pp. 161–174.

[4] M. Devera and D. Cohen. "HTB Linux queuing discipline". In: (2002). URL: http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm. Manual-user guide.

[5] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. "COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. 2013.

[6] A. Dusia, Y. Yang, and M. Taufer. "Network Quality of Service in Docker Containers". In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2015, pp. 527–528.

[7] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[8] M. Ji, A. C. Veitch, and J. Wilkes. "Seneca: remote mirroring done write". In: *USENIX Annual Technical Conference, General Track*. 2003, pp. 253–268.

[9] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. "Designing for Disasters". In: *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*. 2004, pp. 59–62.

[10] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. "SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery". In: *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*. 2002.

[11] D. S. R. Witty. *Disaster recovery plans and systems are essential.* 2001. URL: https://www.gartner.com/doc/340749/disaster-recovery-plans-systems-essential.

[12] P. Reisner and L. Ellenberg. "DRBD v8 - Replicated Storage with Shared Disk Semantics". In: *Proceedings of the 12th International Linux System Technology Conference (Linux-Kongress)*. 2005.

[13] T. Wood, E. Cecchet, K. K. Ramakrishnan, P. Shenoy, J. V.D.Merwe, and A. Venkataramani. "Disaster Recovery As a Cloud Service: Economic Benefits & Deployment Challenges". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. 2010.

[14] T. Wood, H. A. Lagar-Cavilla, K. K. Ramakrishnan, P. Shenoy, and J. V.D.Merwe. "PipeCloud: Using Causality to Overcome Speed-of-light Delays in Cloud-based Disaster Recovery". In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SoCC)*. 2011.