Context-Aware Cloud Topology
Optimisation and Simulation

Predictive Cloud Application Model

Project Deliverable D3.2

Ahmed Ali-Eldin, P-O Östberg, Jakub Krzywda (UMU),
Christopher Hauser, Jörg Domaschka (UULM),
Henning Groenda (FZI)

Due date: 31/12/2014
Delivery date: 27/02/2015

This project is funded by the European Union under grant agreement number no. 610711

Dissemination Level

X	PU	Public
	PP	Restricted to other programme participants (including the Commission Services)
	RE	Restricted to a group specified by the consortium (including the Commission Services)
	CO	Confidential, only for members of the consortium (including the Commission Services)

Version History

Version	Date	Change	Author
0.1	08/12/2014	Initial Document Structure	Ahmed Ali-Eldin (UmU)
0.2	08/12/2014	Executive summary	Ahmed Ali-Eldin (UmU)
0.3	10/12/2014	Video-on-Demand workload model	Ahmed Ali-Eldin (UmU)
0.4	10/12/2014	Wikipedia predictive workload model	Ahmed Ali-Eldin (UmU)
0.5	11/12/2014	Burstiness workload model	Ahmed Ali-Eldin (UmU)
0.6	11/12/2014	Document structure and formatting	P-O Östberg (UmU)
0.7	14/12/2014	Internal review	Henning Groenda (FZI)
0.8	27/12/2015	Introduction	P-O Östberg (UmU)
0.9	28/12/2015	Proactive infrastructure optimization description	P-O Östberg (UmU)
0.91	29/12/2015	Component model descriptions	P-O Östberg (UmU)
0.92	30/12/2015	Scientific computing use case description	Christopher Hauser (UULM)
0.93	31/12/2015	Application model descriptions	P-O Östberg (UmU)
0.94	3/2/2015	Prediction model descriptions and restructuring	P-O Östberg (UmU)
0.95	4/2/2015	Restructuring workload model sections	P-O Östberg (UmU)
0.96	5/2/2015	Internal review	Jörg Domaschka (UULM)
0.97	5/2/2015	Related work	Jakub Krzywda (UmU)
0.98	5/2/2015	Workload modeling	P-O Östberg (UmU)
0.99	9/2/2015	Workload modeling intro	P-O Östberg (UmU)
0.99.1	10/2/2015	Workload modeling restructuring	Ahmed Ali-Eldin (UmU)
0.99.2	17/2/2015	Internal review	Papazachos Zafeirios (QUB)
0.99.3	19/2/2015	Addressing review comments	P-O Östberg (UmU)
0.99.4	26/2/2015	Final polishing	Ahmed Ali-Eldin (UmU)

ABBREVIATIONS

Abbreviation	Description
CACTOS	Context-Aware Cloud Topology Optimization and Simulation
VM	Virtual Machine
QoS	Quality of Service
API	Application Programming Interface
POJO	Plain Old Java Object
VoD	Video on Demand
IaaS	Infrastructure-as-a-Service
ARIMA	Autoregressive Integrated Moving Average
KPI	Key Performance Indicator

EXECUTIVE SUMMARY

This document outlines a framework for the cloud workload and application models used in CactoOpt, the CACTOS infrastructure optimisation tool, and presents initial prototypes for cloud application behaviour models. The purpose of this deliverable is to demonstrate some of the prediction models built for different cloud workloads, and illustrate how they are integrated with the application and component models used in infrastructure and workload deployment optimization. For prediction modelling we give special focus to cloud application user behaviour modelling, including, e.g., workload burstiness and request arrival pattern modelling. To place this work in context, we also present a framework for application and infrastructure modelling focused on translation of workload and application behaviour to infrastructure load.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

Abbreviation	Description
CACTOS	Context-Aware Cloud Topology Optimization and Simulation
VM	Virtual Machine
QoS	Quality of Service
API	Application Programming Interface
POJO	Plain Old Java Object
VoD	Video on Demand
IaaS	Infrastructure-as-a-Service
ARIMA	Autoregressive Integrated Moving Average
KPI	Key Performance Indicator

I. INTRODUCTION

As has been noted by several academic and industry actors in the field, for example Google's John Wilkes, and Battery Ventures' Adrian Cockroft [8], there is currently a need for improvement of the predictability and resilience of cloud data center management tools. As the field of cloud computing transitions into deployment of more mission critical systems such as power and telecommunications infrastructure in heterogeneous data center environments, greater emphasis must be placed on application QoS, stability and predictability of platforms, and development of more advanced control and optimization of infrastructures.

In line with this trend, the research agenda of CACTOS is designed towards exploration of models and optimization mechanisms that consider the broad perspective of multi-objective optimization of data centers, and the models developed and presented in this report are designed to facilitate this work. The end goal of these models is the development of optimization mechanisms that simultaneously target modeling and prediction of load behavior, quantification of load propagation, application and component capacity reservation, and control and optimization mechanisms for proactive and adaptive cloud resource management.

Due to the wide applicability of cloud-based resource provisioning models, Cloud applications span a very wide range of software applications including, e.g., monolithic legacy applications, scientific simulation and data processing applications, distributed tiered applications, and cloud native applications. As such, modeling of cloud applications and their behavior is a complex task that needs to take into account multiple factors such as application composition, deployment configuration, and workload behavior patterns.

In this work we use the following topological definitions: A *cloud application* is a distributed software system where one or more application *components* (software services or subsystems) are deployed in a cloud data center. Components are typically (but not necessarily) deployed in *virtual hosts* using some form of virtualization technology, e.g., hardware supported virtualization (virtual machines), process groups, or software containers, which in turn are mapped onto physical data center (hardware) *resources* using some kind of *deployment constraints*. Deployment constraints constitute rules for the placement and scheduling of virtual hosts on physical resources, and can include, e.g., affinity or anti-affinity constraints that regulate whether or not components can be co-hosted on the same physical resource, or constraints specifying limitations on the amount of hardware resources that can be assigned to components.

In this setting, the main task of CactoOpt (infrastructure optimization) becomes the task of controlling the deployment (placement and horizontal elasticity of applications) and capacity / resource assignments (vertical elasticity) of cloud applications and components so that infrastructure resources maintain acceptable levels of resource utilization while meeting application performance and quality of service (QoS) requirements. To achieve this, it is very important to construct models that accurately capture

application behavior, i.e. how applications respond to external events (e.g., unexpected changes in incoming request patterns) as well as the load applications place on infrastructure resources.

Towards this goal, we define a framework composed of three types of models that combined provide the information required to formulate high-level objective functions for infrastructure optimization:

1. Workload *prediction models* capture the characteristics and variations of application user behavior and incoming request patterns.
2. *Application models* describe the deployment and configuration information of applications in terms of component relationships.
3. *Component models* define the relationships between incoming requests and internal load as well as outgoing requests at component level.

This framework aims to facilitate modeling and translation of application component resource requirements, and structure these in a way so they can be used in the infrastructure optimization tools of CactoOpt. In particular focus in this work are the CACTOS application behavior prediction models that, based on workload analysis techniques, lend predictive power to CactoOpt's modeling of application load.

The remainder of this document is structured as follows: to give perspective on this work, Section II gives an introduction to the type of cloud application models used in CACTOS. After this, Section III goes into details about application behavior prediction models and their use in this type of modeling, and Section IV provides a brief survey of related resource management approaches.

II. PREDICTIVE CLOUD APPLICATION MODELLING

The purpose of the CACTOS cloud application modeling work is to produce models that describe application composition and behavior as a means to model and predict the infrastructure resource requirements of cloud application components. The key challenge in such modeling is to find model representations that preserve the desired qualities to be modeled, capture behavioral patterns accurately enough to provide predictive power in the models, and also provide a structure that is understandable and lends itself to interpretation in modeling. In highly heterogeneous environments such as cloud environments, this proves highly challenging and typically requires composite models that are constructed using multiple types of models that individually capture different parts of systems.

The perspective of prediction taken in this work is two-fold: prediction is based on both behavior modeling and simulation-based experimentation. For behavior modeling we employ statistical and time series analysis methods on both component and application level to build prediction models for different aspects of application behavior. On component level this translates to analysis of component behavior in internal load dimensions such as CPU and I/O usage patterns, and on application level we focus on user behavior and request patterns to identify and isolate trends (and other facets of predictability) in application behavior. To complement and support this approach, we also place focus on construction of model compositions that lend themselves well to computationally efficient simulation-based experimentation. In this work we use simulation for, e.g., in situ evaluation of optimization strategies (testing different simulation strategies as part of the optimization process) and semi-interactive scenario evaluation (simulation of “what if” scenarios). As such, this combined approach allows for inclusion of both prediction and evaluation in optimization: workload prediction and propagation modeling techniques can be used to predict the load of individual components, and simulation techniques can then be used to evaluate the impact of alternative optimization strategies, e.g., to select the optimal virtual machine migration destination based on simulation-based evaluation of how the predicted virtual machine load would interact with existing load on potential migration destinations (physical machines).

To simultaneously capture application behavior and the impact application component load has on infrastructure resources, we here combine three types of models that interlinked model the main interactions of applications and resources in cloud data centers: prediction models, application models, and component models. To give perspective on why this type and level of modeling is used in CACTOS, we here give a brief introduction to the type of infrastructure optimization that is targeted in this work, as well as an overview of each type of model used, before going into detail about the prediction models.

PROACTIVE INFRASTRUCTURE OPTIMIZATION

Cloud infrastructure optimization, here defined as automated infrastructure mechanisms that regulate the provisioning of compute resources to applications in an optimized way, is a complex task. Such

optimization is concerned not only with maintaining optimized levels of resource utilization in data centers, but also in meeting or optimizing the QoS levels of applications. As such, infrastructure optimization a multi-objective optimization problem that requires knowledge and modeling of application composition and behavior, as well detailed modeling of the targeted cloud data center infrastructure.

Proactive infrastructure optimization relies on resource management approaches that use prediction models to extrapolate from prior application or component behavior to predict (a finite time window of) future resource utilization levels[1]. This type of prediction can be done on multiple levels, ranging from application request patterns to internal resource (e.g., CPU) utilization patterns for individual components. The models used in this work are designed for two primary (optimization-related) purposes: to establish upper and lower bounds of the hardware resource assignments of application components (capacity bounds that are used in placement and scheduling of components), and to within these bounds accurately model and predict variations in resource usage patterns.

The interpretation of the modeled upper and lower capacity bounds used here is the following: upper bounds of component capacity requirements define the maximum capacity (in some internal load dimension such as CPU, RAM, I/O, or storage) a component will need within a foreseeable time window, which thus defines a maximum size of a virtualized host (e.g., virtual machine or software container) for that component. Similarly, lower bounds of component capacity requirements defines the minimum capacity a component needs to be able to deliver some aspect of the desired QoS of the application. As the upper and lower bounds respectively represent a maximum and minimum size of a virtual host, they combined contain all the information needed to perform (coarse-grained) placement of virtual hosts. In addition, component resource capacity bounds also provide intuitive interpretations of how much (minimum) capacity needs to be reserved for a virtual host, as well as of how much spare capacity a physical resource has left (after the maximum capacity of the virtual hosts are allocated) for other components, and are thus suitable to be used as coarse-grained deployment constraints for virtual hosts.

Within the upper and lower bound, actual resource consumption is tracked and modeled for prediction. This can as stated be done for individual components by directly tracking component performance in individual resource capacity dimensions, e.g., identify cyclic patterns or phases in execution patterns (as done in the Molpro case where CPU and I/O phases are modeled to predict execution phases and completion times), or by modeling the load propagation between application components in application models (further discussed the application model section). Regardless of method, the aim of this modeling is the same: to predict the load an application component places on its (virtual or) physical host to enable higher resource utilization or application QoS through optimization of deployment and configuration of components. Optimization approaches that utilize this kind of information range from, e.g., scheduler optimization approaches such as , overbooking approaches such as[28, 26], and adaptive controllers that regulate application QoS after available capacity[18]. In this modeling, formerly established upper and lower bound of capacity can also be used as triggers for detection of prediction errors. If a component consistently uses less than its lower bound allocated capacity it may indicate that the lower bound is set too high, and conversely if a component is repeatedly using all of its allocated capacity (i.e. hits its upper bound), it may indicate an opportunity to spawn additional instances of that component (horizontal elasticity).

Taken together, these types of modeling facets allow optimization mechanisms to address both spatial and temporal variations in load. Spatial variations (e.g., differences in sizes of virtual machines) can be addressed in placement optimization using the upper and lower bounds for component capacity requirements. Temporal variations (i.e. variations over time in the load components place on their hosts) can similarly be addressed using adaptive vertical elasticity optimization (control) routines that operate on the load predictions of individual components. Similarly, the predicted load patterns of components (combined with the application configuration information of the application models) also allows formulation of horizontal elasticity systems where autonomic application monitoring routines can identify application components or segments that are under- or over-utilizing their allocated resources.

COMPONENT MODELS

As mentioned in the previous section, coarse-grained scheduling and initial placement of (virtual hosts of) components can be done using upper and lower bounds for capacity requirements. For adaptive control and prediction of resource use however, more fine-grained modeling of the translation between incoming request patterns and internal as well as outgoing load is needed. In this work we define a component model designed to quantify the relationships between incoming request patterns and a) internal load in the dimensions of CPU, RAM, I/O, and storage; and b) outgoing request patterns. In this modeling we further make the observation that for each of these modeled entities there are significant differences between different types of applications in their load patterns. There are for example some applications that are load-wise driven directly by external requests, e.g., web servers where the needed resource capacity directly correlates to the type and amount of incoming requests (web servers are mostly idle when not processing HTTP requests), and other types of applications that are primarily driven by internal load factors, e.g., batch-oriented scientific processes that receive a few requests and then spend large amounts of time and resources performing computations that are not directly correlated to the incoming request patterns in any externally visible way. For this reason we also further decompose our component model to encompass the notion of foreground (load directly driven by incoming requests) and background (load not driven directly by incoming requests) load.

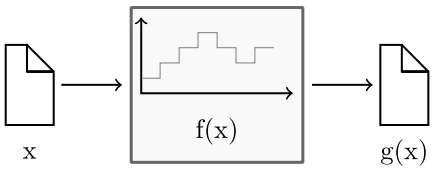


Figure 1 Load can be driven by incoming requests or by background jobs.

It is worth noting that upper and lower component capacity bounds can (for known applications) be set at deployment time, either explicitly by experienced system administrators or even implicitly by, e.g., inferring them from the QoS level a customer at a public cloud offering is paying for. A common pattern in

public IaaS cloud offerings is for example to distinguish between different service levels where a certain price level corresponds to a certain "size" of a virtual machine, e.g., the Amazon Web Service (AWS) C4 instances where a "c4.large" virtual host has (as of early 2015) 2 virtual CPUs and 3.75GB RAM, while a "c4.xlarge" instance has 4 virtual CPUs and 7.5GB RAM, etc. These limits can also be set by benchmarking applications under representative settings, or be trained using machine learning approaches operating on historical usage information (when available). Naturally, such controls can also be assigned and controlled in run-time using similar approaches [29, 16].

These models must capture both the direct and indirect (e.g., co-hosting overhead and noise) relationships between incoming and propagated load, and optimization algorithms must also consider the effect this load has on component environments. For this reason, we here consider a type of grey-box modeling where component load is monitored in several dimensions (CPU, RAM, I/O, storage) at multiple levels (incoming and outgoing requests, load within components, load at virtual host level, as well as load at physical host level).

As with all modeling approaches, there exists several trade-offs between the resolution and accuracy of the model and the predictive power and computational efficiency of the model. In this approach we are purposefully selecting a coarse-grained modeling approach that in its initial versions makes several simplifications, of which the most obvious probably is the assumption of homogeneity of incoming requests. We select this level of modeling to achieve high computational efficiency in order for the model to scale large data centers and still be useful in online simulations of the model. In later versions of the model we also intend to study how statistical modeling of non-homogenous request patterns can be incorporated in this model, as well as techniques for automatic training and parameterization of the model for different applications.

APPLICATION MODELS

For monolithic legacy applications that execute in a single virtual host, modeling of component behavior is enough to describe application behavior. For distributed systems such as cloud applications composed by multiple components deployed on different physical hosts however, some kind of modeling of the interrelationships and data flows between components is needed. To complement the component model described in the previous section, we here define a graph-based application model intended to capture the structure, configuration, and hierarchy of distributed applications.

The overall purpose of this modeling is to quantify the propagation of load between components in applications, i.e. to build graph-based translation functions that allow modeling and quantification of the load incoming requests place on different components within an application. By viewing applications as sets of components that are linked over networks, and using the load propagation functions of the components in combination with the application graph link data, we construct models that allow quantification of both background and request-driven load placed on individual components over time.

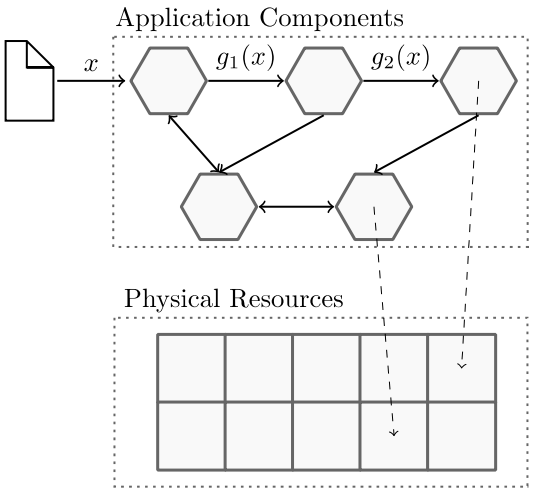


Figure 2 Applications can be modeled as graphs

As illustrated in Figure 2, applications are viewed as (potentially cyclic) directed graphs of software components where graph nodes model components, graph links indicate coupling between components, and link weights indicate load distribution patterns. Making the assumption that load propagation for request-driven load is instantaneous (e.g., occurs within a single time step of a discrete time simulation), application-level load propagation functions are derived from component load propagation functions and the graph link data.

Central to cloud application deployment and management are application description templates that outline configuration and parameterization of components, e.g., OpenStack Heat, AWS CloudFormation, and Google deployment templates. As the purpose of these deployment descriptors is to enable management of applications (rather than individual virtual hosts), they naturally contain much of the graph link information needed in the CACTOS application models. Lacking such deployment descriptors, we envision that application models can also be trained using machine learning techniques based on network monitoring data (a topic for future study in CACTOS)[9].

Natural extensions to this model include, e.g., network modeling that (similar to the component resource capacity bounds estimation) quantify bandwidth capacity and link quality requirements between components. In infrastructure optimization, it is also envisioned that horizontal elasticity mechanisms can be effectively realized as autonomic components operating on application models, e.g., using monitoring data in combination with predicted upper and lower capacity bounds for components as triggers for horizontal elasticity adjustments (i.e. removal or instantiation of component instances).

PREDICTION MODELS

Given application and component models that capture the interrelationships of cloud application components and the component-wise translation functions between incoming requests and (internal and external) load, it is possible to (based on the incoming request patterns of an application) formulate functions that in detail estimate the resource requirements and load impact of applications in cloud data center. Extending this capability with prediction models then formulates the basis of a framework for combined application and infrastructure modeling for infrastructure optimization. What the CactoOpt cloud application models aims to realize is exactly this, a model framework for prediction of application and component load behavior that can be used in both simulation and runtime environments.

Prediction of application behavior is often done based on application workload pattern analysis as application characteristics such as user behavior and seasonal load patterns are preserved at this level. In this work we focus on statistical and time series based analysis of historical application behavior, e.g., analysis of request logs or workload traces to identify seasonal and trend patterns, to allow fine-grained modeling of application and user behavior that can be used in prediction of future workload patterns. As demonstrated in the following sections, such analysis for distributed cloud applications can be greatly beneficial in building knowledge of the structure and elasticity patterns of applications.

For applications where the internal load is not primarily driven by external requests, e.g., monolithic batch processing applications, modeling of application request patterns is not very illustrative. For such applications we note that behavior prediction can be performed using similar time series analysis techniques on component level (i.e. looking directly at the resource usage patterns rather than the incoming request patterns). While use of application behavior knowledge can greatly assist in this process, this type of modeling can be done using a black-box perspective (i.e. without knowledge of the application internals) for, e.g., application classification for cloud autoscaling [4]. This technique is demonstrated in the scientific computing use case where we illustrate how phase analysis can be used to predict execution times and provide scheduling information for the Molpro application.

III. APPLICATION BEHAVIOR MODELING

The performance of cloud applications can be considered as a function of three main factors: the design of the system, the implementation of the system, and the load on the system. When a system is designed, there is typically an underlying assumption of the system's operational load range. However, the actual workload, and thus the performance, is only known when the system becomes operational, which can sometimes turn system design and workload analysis into a chicken-and-egg problem. A common approach to get around this issue is to analyze workloads of existing (similar) systems and draw similarities and possible discrepancies between the existing system and the new system.

In this work we demonstrate how predictive workload models are developed and used in conjunction with (application and component) load propagation models for infrastructure optimization in the CACTOS infrastructure toolkits. The workload models are constructed using workloads acquired from project partners as well as publicly available workloads, and are used to understand the way (users of applications and) applications behave and the effect workload events (such as planned and unexpected workload peaks) have on infrastructure resources.

The workload models presented here are based on six different workloads. The UULM workload represents an HPC application with alternating periods of I/O intensive and CPU intensive loads (Molpro) [31]. The second workload is a Video on Demand (VoD) application obtained from a major Swedish VoD provider [2]. The third workload is a 6 years' workload obtained from the Wikimedia foundation for all the services running on their servers in all languages including Wikipedia, Wikibooks and Wiktionary [10]. For burstiness modeling we have also used traces from IR-Cache, a caching service consisting of approximately ten caching proxies located throughout the United States, a Google cluster workload, and a workload from the FIFA 1998 World Cup servers[3]. More information about the workloads used in this work can be found in Appendix A.

MODELING USER BEHAVIOUR IN CLOUD APPLICATIONS

One of the most direct ways to model user behaviour in cloud applications is to study cyclic and seasonal patterns in application request workloads, the distributions which generate the workload and the popularity of different objects between the users. To build understanding and models of user behaviour in these contexts we here study request arrival patterns in the Wikipedia workload, one of the largest publically available workload traces, a VoD workload from a Swedish VoD provider and a workload from the MolPro application.

ARRIVALS IN THE WIKIMEDIA WORKLOAD

We used time-series analysis to model the hourly workload request flow. In the classical decomposition model [9], an observed time series $\{x_t, t \in T\}$ is modelled as a realization of a sequence of random variables $\{X_t, t \in T\}$, $X_t = T_t + S_t + R_t$, where T_t is a slowly changing function (the trend component), S_t is a periodic function with known period *m* (the seasonal component), and R_t is a random noise component (assumed to be a stationary time series). For modelling the request flow, we use (seasonal) Autoregressive Integrated Moving Average (ARIMA) models.

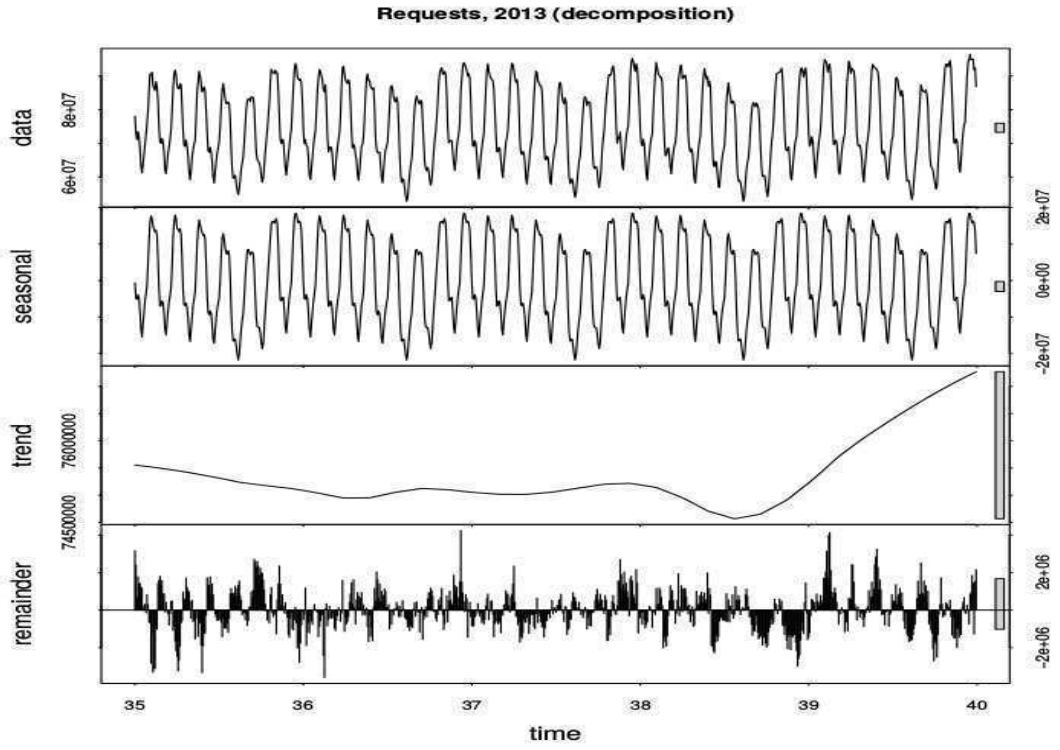


Figure 3 Time series decomposition of the hourly arrival rates for year 2013

The workload is clearly non-stationary due to changing mean of the arrival rate with time. Therefore, instead of building a single global model, we modelled smaller periods of the workload where there was no significant step. Figure 3 shows the decomposition of the request flow for the period between the 3rd of September till the 8th of October, 2013. The seasonality graph suggests high daily and weekly correlations between the number of requests during this period. The trend graph can be approximated with piecewise-linear functions. We fitted the remainder to a seasonal ARIMA (2, 0, 2) (0, 0, 1) model using the R forecast package.

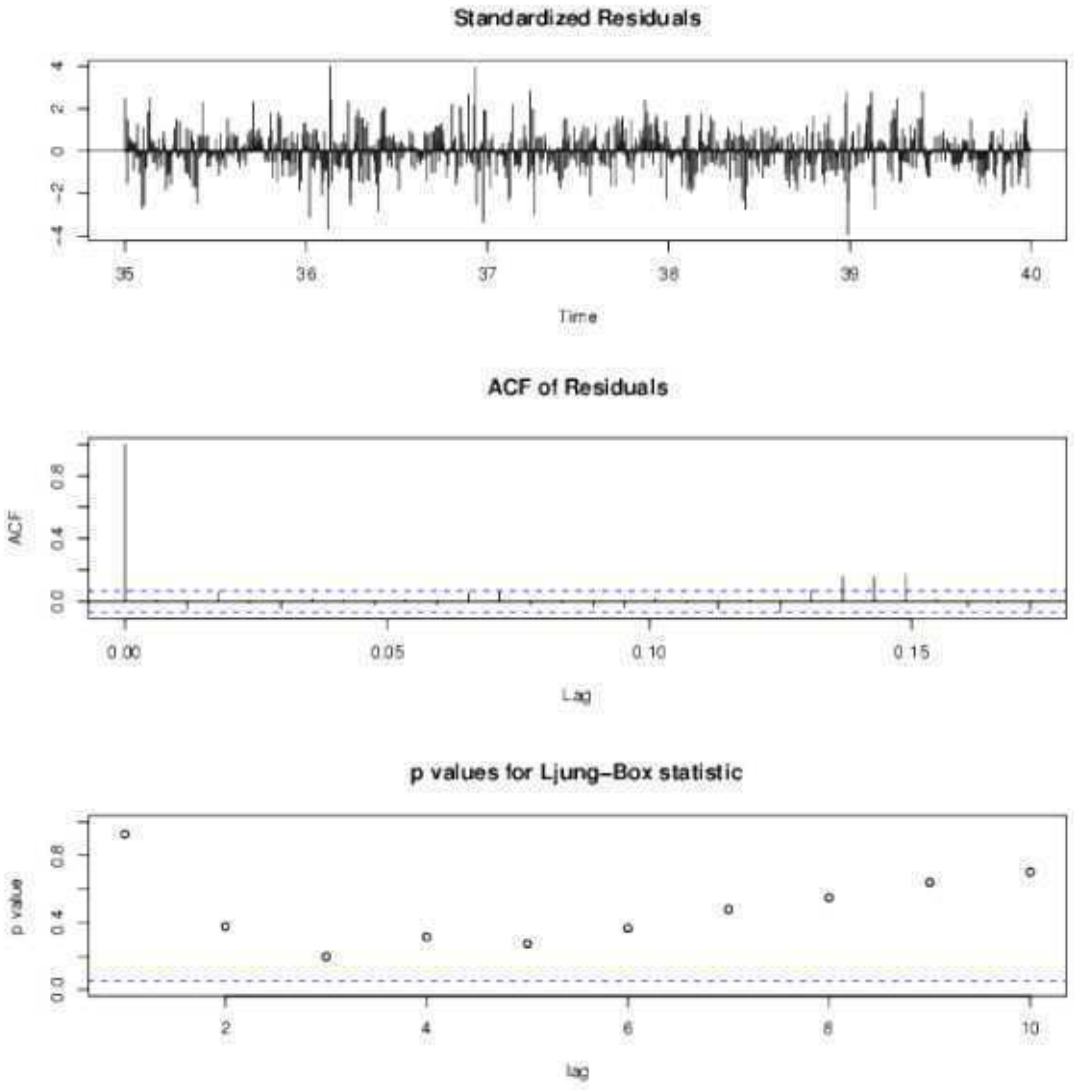


Figure 4 The residuals are tested in order to find the quality of the fitted time-series model.

Figure 4 shows some diagnostic plots of the fit, namely, the standardized residuals, autocorrelation for residuals, and p-values for Ljung-Box statistics for the fitted model [9]. While most of the standardized residuals are well inside the [-2, 2] interval, in some cases they are outside. The autocorrelation function (ACF) also suggests some daily correlation still present in the residuals, so the fitted model could still be improved. We leave this for future work.

We have repeated similar analysis for different (regular) parts of the workload and summarized our results in Table 1. The first column is the period studied. The second column summarizes the trend characteristics. For most parts of the workload, the variance in the trend is less than 5% of the average value. The second

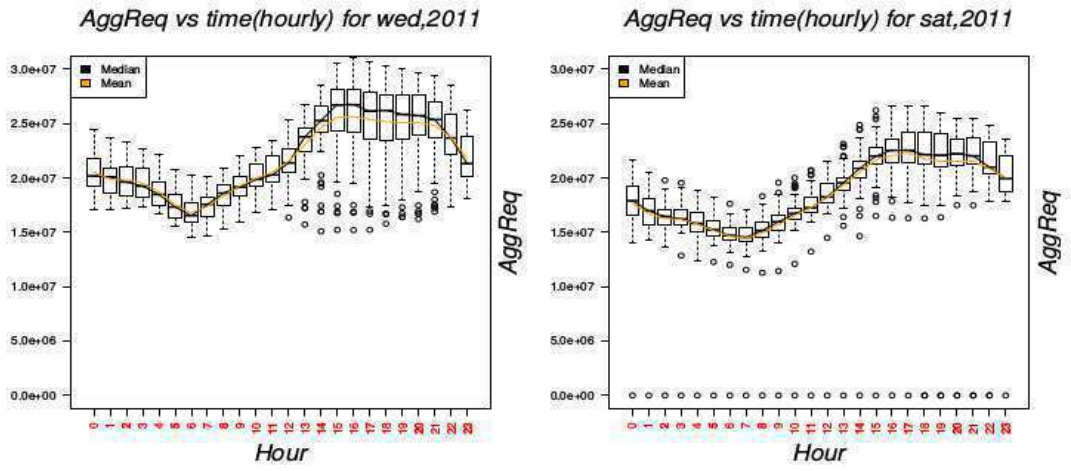
column summarizes the workload seasonality. For each seasonality plot, we have computed the FFT. The only two dominant frequencies are at 1 day and 1 week. Their amplitudes are also noted. The last column describes the fits for the remainders. The remainders are fitted using seasonal ARIMA models. We note that the models are of low order and are not too far from each other with even some models repeating, e.g., ARIMA (1,0,2)(0,0,1) is repeated 4 times.

Period (y, d/m)	Trend min, mean, max (10^6)	Seasonal periods/amplitudes (days)/(10^6)	Remainder ARIMA[168]
2008, 17/05-21/06	15.01, 15.14, 15.28	1.00/6.57, 6.74/2.89	(1,0,2)(0,0,1)
2008, 31/07-04/09	12.49, 13.36, 13.86	1.00/4.40, 7.26/2.35	(1,0,2)(0,0,1)
2008, 26/10-30/11	14.58, 15.22, 15.63	1.00/7.57, 7.26/2.14	(1,0,0)(1,0,0)
2009, 09/01-13/02	15.42, 16.13, 16.33	1.00/7.84, 6.74/2.26	(1,0,0)(1,0,1)
2009, 23/04-28/05	15.68, 16.28, 16.89	1.01/7.52, 7.26/2.90	(1,0,2)(0,0,1)
2009, 17/05-21/06	15.40, 16.10, 16.50	0.99/6.56, 7.14/2.87	(2,0,1)(0,0,1)
2010, 28/07-01/09	18.50, 19.08, 20.01	1.01/6.67, 7.26/3.03	(1,0,2)(0,0,1)
2010, 20/08-24/09	19.24, 19.87, 20.63	1.00/8.18, 6.85/3.14	(2,0,2)(1,0,1)
2011, 01/01-04/02	18.73, 21.34, 22.19	1.00/10.20, 6.74/2.53	(3,0,3)(1,0,1)
2011, 20/06-25/07	19.34, 19.75, 20.39	1.00/6.90, 7.26/2.44	(1,0,3)(0,0,0)
2011, 01/11-06/12	22.98, 23.35, 23.78	1.01/9.71, 7.26/2.94	(1,0,0)(1,0,0)
2012, 01/01-04/02	23.01, 25.15, 25.85	1.01/10.14, 6.74/2.05	(3,0,3)(0,0,0)
2012, 07/03-11/04	23.09, 23.73, 25.23	1.01/9.62, 7.26/2.81	(1,0,0)(1,0,1)
2012, 25/08-29/09	25.85, 26.56, 27.23	1.00/9.39, 6.74/2.89	((1,0,1)(1,0,1)
2013, 19/01-23/02	65.87, 66.83, 67.79	1.01/33.96, 6.74/9.77	(1,0,2)(1,0,1)
2013, 15/04-20/05	60.09, 61.07, 64.24	1.01/30.09, 6.74/8.81	(1,0,0)(0,0,1)
2013, 03/09-08/10	74.57, 75.33, 77.27	0.99/26.56, 6.93/6.56	(2,0,2)(0,0,1)

Table 1 Different time-series fits are obtained for different parts of the data

To understand the differences between the load on the servers during weekends vs. weekdays, Figure 5 shows box-plots for the aggregate hourly number of requests for all Wednesdays in 2011 and all Saturdays in 2011. A box-plot is a way to visualize the quartiles and the dispersion of the distributions of the data [19]. The medians and the means for the different hours are plotted. It is clear from the figure that the time of the day affects the number of requests significantly with the lowest activity at 5, 6 and 7 a.m. and the highest activity during the afternoons and the nights. The data dispersion is also affected by the time of the day and the day of the week. More accurate predictions can be done at times with lower dispersion.

Similar results were obtained for different days and years for both the number of requests and the sent data.



(a) Time of the day for Wednesdays vs aggregate request distribution.

(b) Time of the day for Saturdays vs aggregate request distribution.

Figure 4 Box plots of the hourly arrival rates on all 2011 Wednesdays and Saturdays

BURSTS IN THE WIKIMEDIA WORKLOAD

We now shift our focus towards studying the effect of major events on the workload. We focus on two major events, Michael Jackson's death and the Super bowl XLV. For both events, we report results for the collateral load that accompanied the load on the main page. We define the collateral load as the load increase on pages associated with the page of the main event excluding the load on the page dedicated to that event. In order to extract the collateral load, we parse the main event's page for the webpages that it links to. Figure 6 shows the workload on Michael Jackson's page and the collateral load. When Michael Jackson died, the load on his page increased by around four orders of magnitude. This increase was accompanied by an increase in the load on all pages that his page linked to, but with a smaller yet significant amplitude. Both the load and the collateral load started decreasing shortly after a few hours but with the collateral load decreasing slower than the load. After 12 days, on the 7th of July, Michael Jackson's memorial service took place, resulting in another significant load spike on the load on his Wikipedia entry, but in a much larger spike in the collateral load.

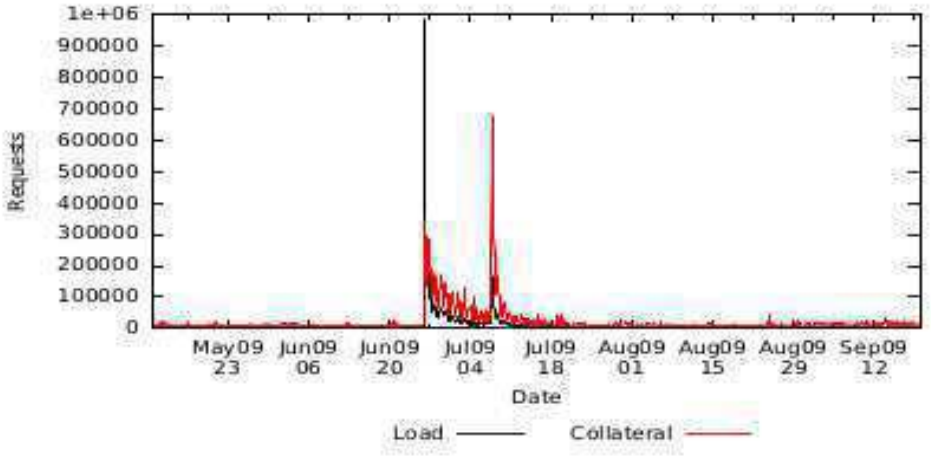


Figure 5 When Michael Jackson died, the traffic to his Wikipedia page increased by three orders of magnitude. The traffic on pages linked from his page also increased by more than two orders of magnitude.

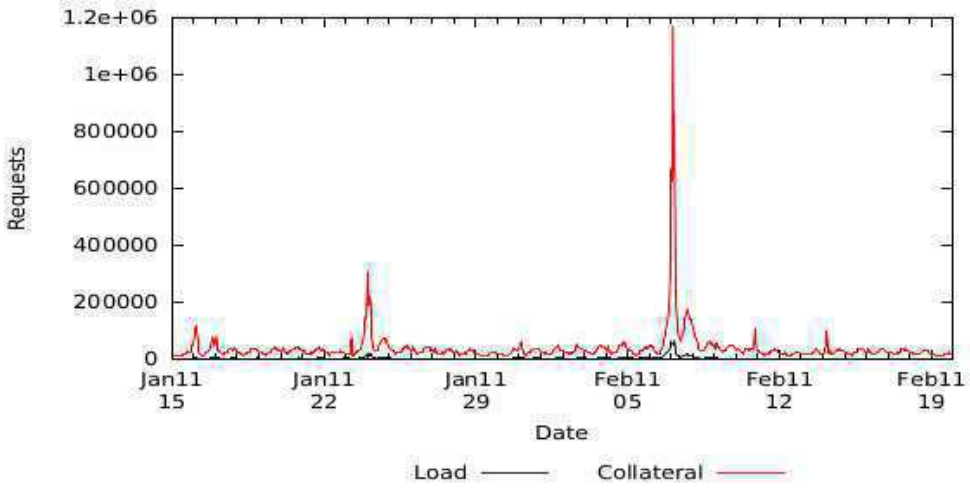


Figure 6 The Super Bowl causes a yearly spike that occurs on pages related to the event but not the main event's Wikipedia page

Figure 7 shows the load on the Super Bowl XLV page and the collateral workload on the Wikimedia servers before and after the event. Although the main event was the Super Bowl, the load spike was in the collateral workload dwarfing the spike on the Super Bowl page. We obtained similar results for the FIFA 2010 World Cup, the Eurovision 2010 and the Egyptian revolution articles where for all of them the collateral workload was typically orders of magnitude than the load on the main article. This phenomenon seems to be common for planned events, where the collateral load surpasses the load on the original item. We plan to study the collateral load effect in more details and find its implication on resource management problems such as placement and resource provisioning.

WIKIMEDIA CONTENT POPULARITY MODELING

During peak hours, less than one third of Wikimedia's pages are accessed while during hours of lower activity around one sixth of all pages are accessed. This pattern did not change for the period of study. To better understand the dynamics of the workload on individual pages, we keep track of the top 500 pages having highest number of requests for the period of study. This is a highly dynamic list with over 650000 pages joining the list for some time during the period of the study. The least accessed page in the top 500 list had on average less than 1200 page views per hour for the period of the study, i.e. less than 20 page views per minute. On the other hand, the most popular pages in the list where usually general pages, e.g., the English Main Page. These pages had on average more than 500000 page views at the beginning of our study and around 20 Million views at the end.

Figure 8 shows the histogram of the number of consecutive hours a page stays popular before leaving the list. Most pages have volatile popularity, with 41.58% of the top 500 pages joining and leaving the top 500 list every hour, 87.7% of them staying in the top 500 list for 24 hours or less and 95.24% of the top-pages staying in the top 500 list for a week or less. The distribution has a long tail. Since there are some monitoring gaps in the workload, we were not able to infer which pages were in the top 500 list during these gaps. This adds some uncertainty to the preceding results. In order to reduce this uncertainty, we plot Figure 9 that shows the percentage of time a page is in the top list during the study period. The x-axis represents the ratio between the total time an object stays in the top 500 list and the total time of the study while the y-axis shows the frequency of objects with a certain ratio. Around 9 pages where in the top 500 list for the whole period of the study, these are mostly the main pages for different Wikipedia projects. On the other hand, Figure 8 confirms that most objects are popular for only short periods of time.

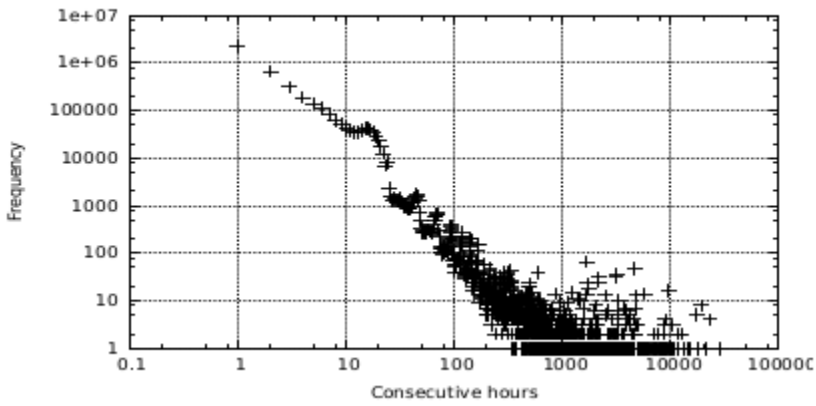


Figure 7 Time a page stays in the top 500 popular Wikipedia articles

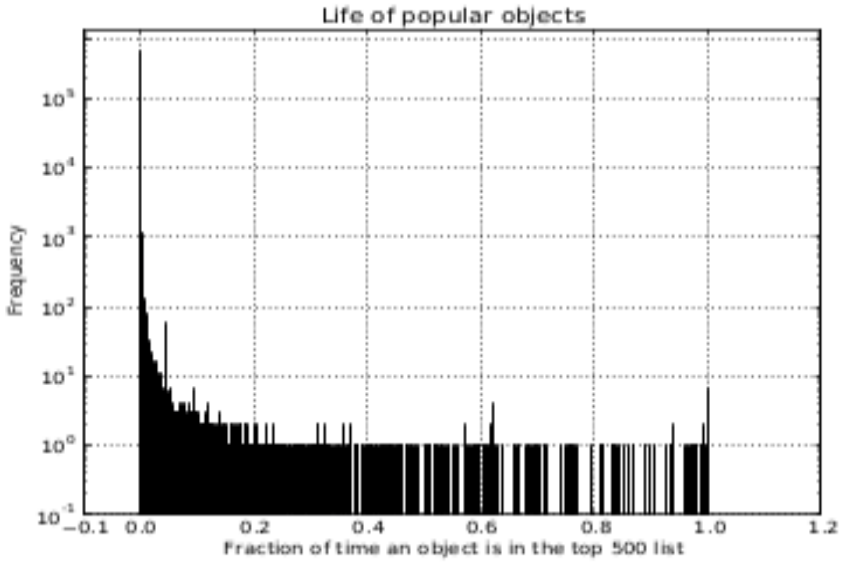
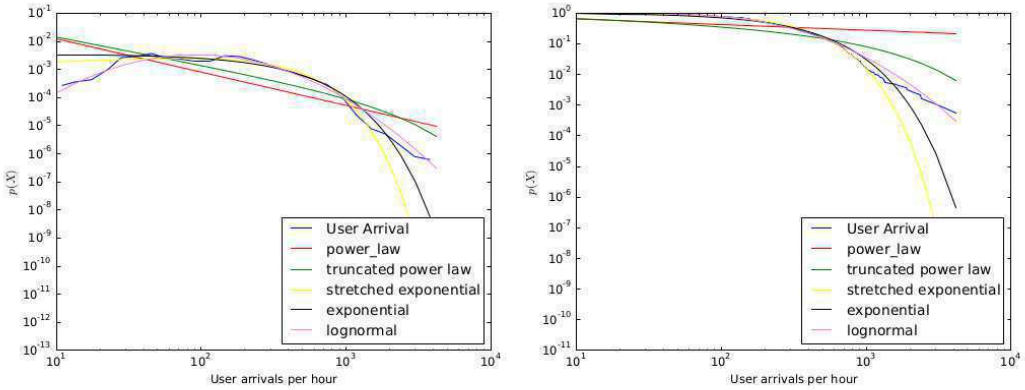


Figure 8 Ratio between the time a Wikipedia article stays among the most popular 500 articles and the total time of the study

VOD REQUEST ARRIVAL RATE MODELING

The Probability Distribution Function (PDF) and the Cumulative distribution Function (CDF) of the hourly session arrival rate is shown in Figure 10 (in blue) on a Log-Log plot. An almost identical plot was also obtained for the user arrival rate since one user almost always does not start more than one session per hour. The PDF suggests that the arrival rate process can be modelled using a heavy tailed distribution. We have fitted the arrival rate data to different distributions and compared the goodness of fits in order to find a good fit. The data was fitted to lognormal, exponential, truncated power law, stretched exponential, gamma and power law distributions.

The plots show that either a lognormal distribution, an exponential distribution or a stretched exponential distribution is a good fit. To choose the best fit, we used the Kolmogorov-Smirnov (KS) test. The p-value for both the lognormal distribution and the stretched-exponential distribution was greater than 0.05, the least significance level required to validate the null hypothesis that the empirical data does not follow the distribution. To be precise, the KS distance for the lognormal distribution is 0.077 with a p-value of 0.09, and the KS distance for the stretched exponential distribution is 0.059 with a p-value of 0.31.



(a) PDF of the distribution of the arrival rate and different fits.

(b) CCDF of the distribution of the arrival rate and different fits.

Figure 9 Fitting the request arrival rate to different distributions

Both the lognormal and the stretched exponential distributions are possible fits given the p-values of the KS test. To identify the better fit, we use the log-likelihood ratio between the distributions. The log-likelihood ratio of the lognormal distribution was higher with a p-value of 0.01. We thus conclude that the lognormal distribution is the best distribution to fit our data from the distributions tested. The fitted lognormal distribution is different from the arrival rate distribution of the VoD service provided by China Telecom discussed by Yu et al. Where the arrival rates follows a modified Poisson distribution [32].

Figure 11 shows the PDF of the video sessions' inter-arrival times (seconds) on a log-log scale. More than 50% of the sessions start after one or less than one second from the arrival of the previous session and around 90% of the sessions start within a minute from a previous session. The maximum inter-arrival time is around 24 minutes. We have again tried fitting a distribution to the Inter-Arrival time following the same steps described above. Again, the KS test showed that both the lognormal distribution (p-value=0.08 > 0.05) and the stretched exponential distribution (p-value=0.08 > 0.05) to be two viable fits. Testing using the log-likelihood method described by Clauset et al. [7], the stretched-exponential distribution has a higher likelihood than the lognormal distribution with a p-value=0. While it is popular to model session and user arrival rates as Poisson processes in workload generators [25] [14], our results suggest that for different VoD services, different models of arrival might occur. Poisson processes require the inter-arrival time distribution to be exponential. Figure 10 shows also the best exponential distribution fit we could achieve for the inter-arrival data. The deviation clearly shows that the inter-arrival time distribution is not exponential, and thus the arrivals do not follow a Poisson process. Poisson processes were considered the defacto processes to model network arrivals until the seminal work by Paxson and Floyd [20]. It is thus worth investigating if Poisson processes fail also to model arrival processes for VoD systems. We unfortunately do not have sufficient data from enough VoD providers to come to such a conclusion.

Since at least for the TV4 workload, the user and request arrival processes cannot be modelled using Poisson processes, many of the previously developed theories and models based on the assumption of requests/users generated from a Poisson process will either be inaccurate or will be wrong for systems like

TV4. Since VoD workloads are scarce, we cannot compare our results with systems other than the very few available in the literature.

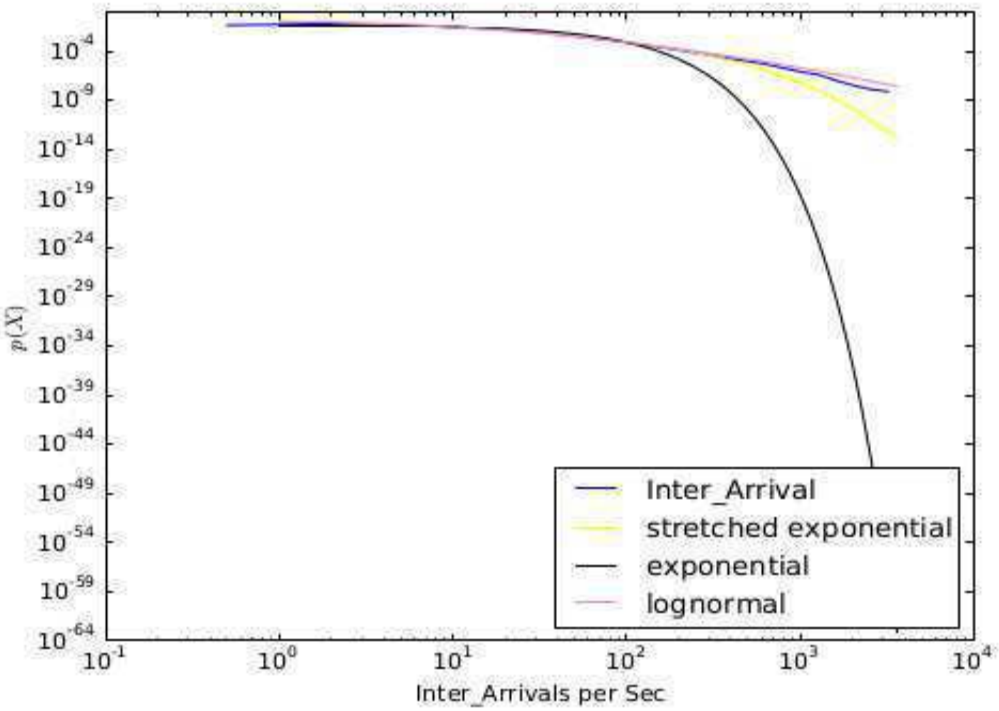
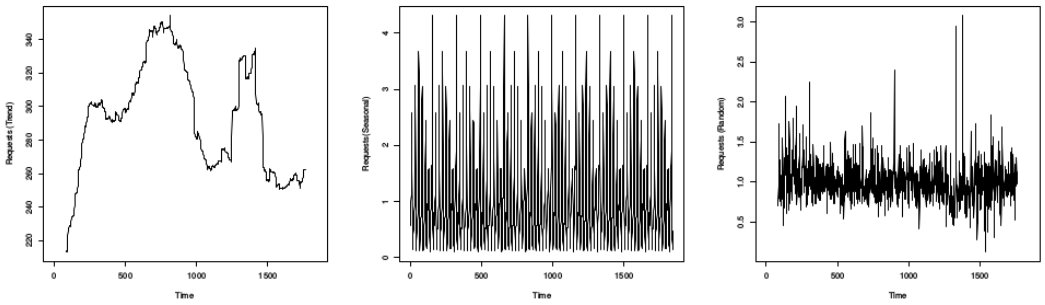


Figure 10 The inter-arrival rate cannot be fitted to an exponential distribution, i.e., the arrival rate does not follow a Poisson process

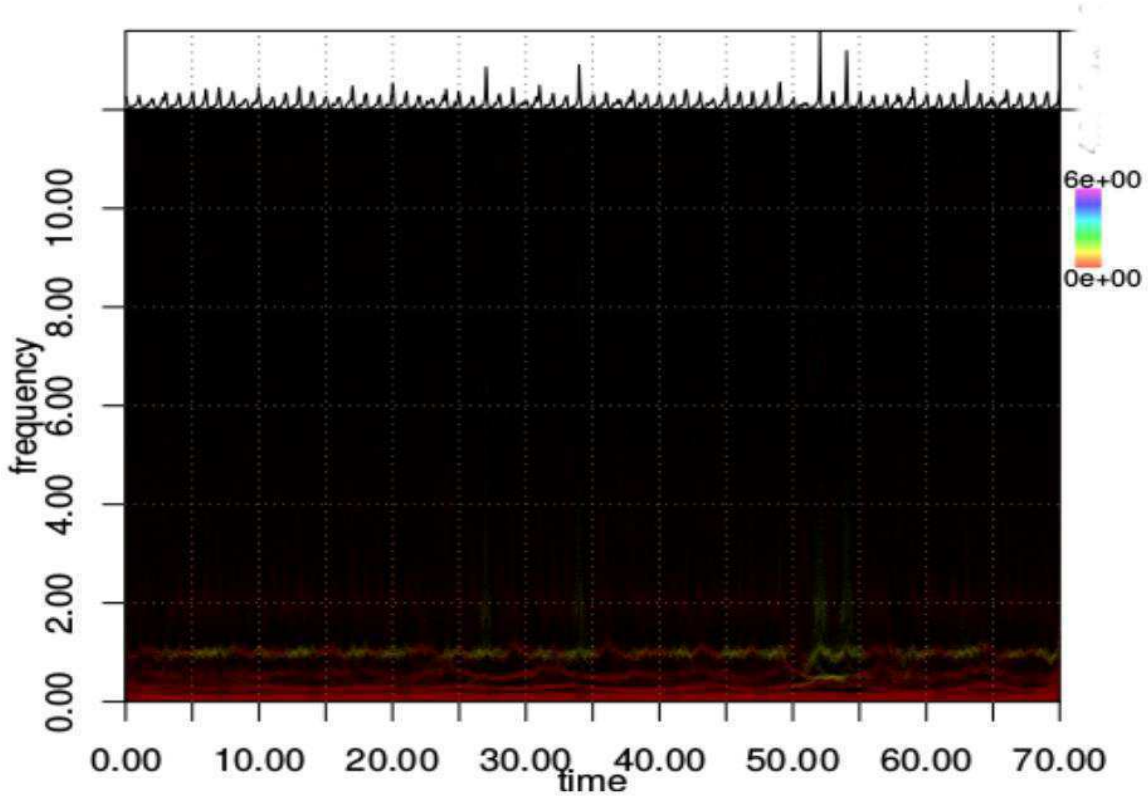
TIME SERIES MODELS FOR VOD REQUEST ARRIVALS



(a) The trend component T. (b) The seasonality component S. (c) The random component r.

Figure 11 Time Series (multiplicative) decomposition of the VoD session arrivals

For the VoD workload, we show in this report another decomposition and modelling technique suitable for non-stationary time-series. The multiplicative decomposition of the request arrival rate is shown in Figure 12. The multiplicative time-series decomposition again decomposes the time-series into three components the trend (T), the seasonality (S) and the random components (r). The difference is instead of decomposing the time-series into additive terms, it is decomposed in to multiplicative terms, $X = T \times S \times r$.



Figur 12 The Hilbert-Huang transform is used to obtain the time-frequency representation of the workload

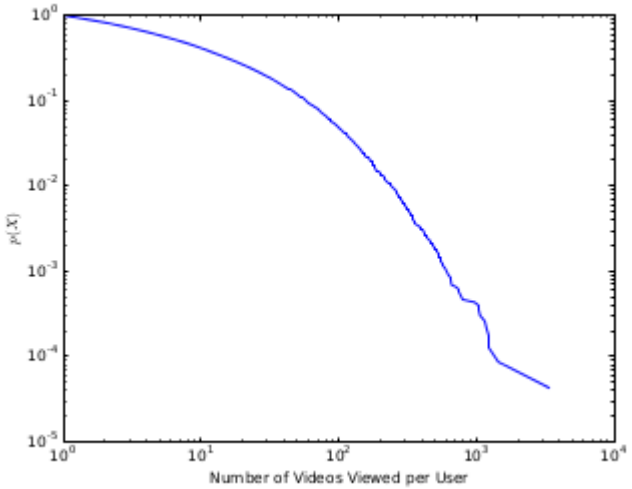
In their seminal work to model non-stationary time series, Huang et al. introduced a novel empirical method to characterize the frequency variations in non-linear and non-stationary time-series [12], recently, known as the Hilbert-Huang Transform (HHT). At the core of the HHT is the Empirical Mode decomposition (EMD) method and its different variations [24].

The EMD (and all its other variations) are methods with which any complicated data set can be decomposed into a finite and often small number of Intrinsic Mode Functions (IMF) that admit well-behaved Hilbert transforms. The Hilbert spectrum can then be used to visualize the produced IMFs and frequency variations in the original signal. Since the number of IMFs produced is low, it is a more efficient way of spectral analysis compared to, for example, the Fourier transform which typically requires an

infinite number of sinusoidal frequencies to represent any time-series. Huang et al. and others have discussed the strengths and weaknesses of their proposed method and showed the superiority of the HHT compared to other available spectral analysis methods such as the wavelet transforms and Fourier transforms [24,12] .

The Hilbert spectrum is shown in Figure 13. The X-axis is the time in days and the Y-axis is the frequency in weeks. The colours represent the intensity of the frequency component at any point in time. On top of the graph, the analysed time-series is plotted. The low frequency components dominate the time-series. The strongest of these components is the weekly component. At the times of the four major spikes in the VoD workload between 25 and 35 days, and 50 and 55 days in Figure 6, the spectral pattern is distorted. The spikes cause an increase in the power and dispersion of the spectrum of the time-series. This suggests that a possible way to detect spikes as they occur would be to use spectral analysis methods to detect the beginning of the spike [3]. We talk later about spike modelling in the report.

VIDEO VIEWS PER USER



Figur 144 Most users do not use the service on daily basis

Figure 14 shows the CCDF of distribution of the number of videos viewed per user. More than 90% of the service users view less than 70 videos during the period of the study, i.e., less than one video per day. Many of these sessions last for less than 10 minutes. To see how long a user uses the VoD service, Figure 15 shows the CCDF of the total time a user used the VoD service. Some users have used the service for just a few minutes, with more than 25% of the users using the service for 45 minutes or less. Other users have used the service heavily. The longest usage was by a customer who used the service for a total of 45 days and a few hours. This can be either a user who has the service running for over 15 hours per day, like a

restaurant using the service, or a customer who has multiple devices all connected to the service using the same ID.

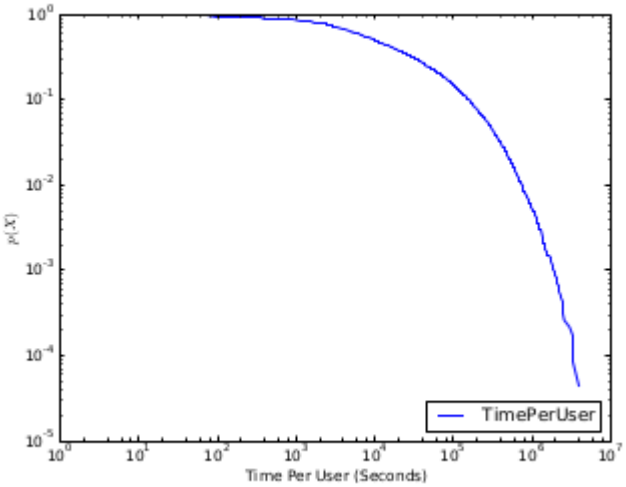
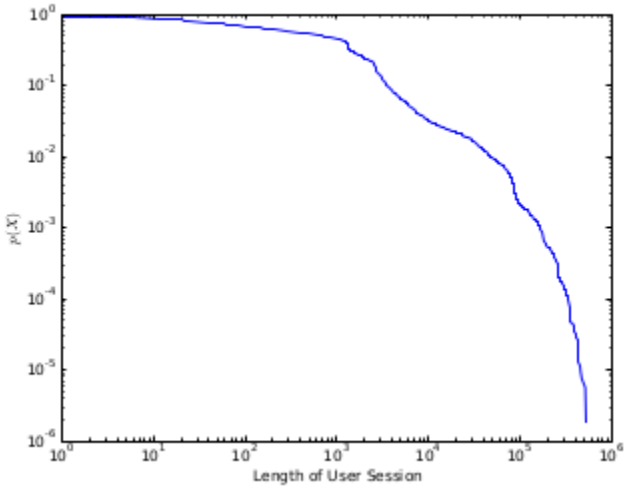


Figure 15 Most users use the service moderately.

IMPATIENT USER BEHAVIOUR



Figur 16 User sessions are quite short

Not all users who start viewing a video stream continue to watch until the end. Some users keep replaying the video, going back and forth in the video and pausing the video. This leads to sessions having very different length. Figure 16 shows the complementary CDF (CCDF) for the length of the VoD sessions. More than 90% of the sessions last for less than one hour, with more than 50% of the total sessions lasting less than 12 minutes. More than 20% of the sessions gets terminated within the first 30 seconds from their start time. These numbers confirm the "impatient user behaviour" discussed in previous studies described by Yu et al. [32]. Although the difference between our study and Yu et al.'s study is around 6 years, the numbers we find here do not differ considerably from their study. For example, Yu et al. found that 50% of the users terminate a session within the first ten minutes from when they start it and that more than 90% of all sessions terminate within 60 minutes from when they start. The main difference between our study and Yu et al.'s study in this respect is that the users of the TV4 VoD are more likely to stay than the users in Yu et al.'s study if they make it past the first 10 minutes.

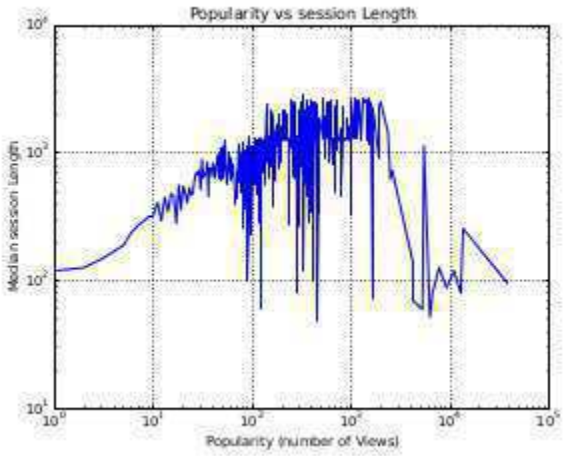


Figure 17 Users tend to be impatient for both extremely popular and extremely unpopular videos

Figure 17 shows how the median session length changes with the popularity of the videos (number of views). For extremely unpopular and extremely popular videos, the "impatient user behaviour" is quite high with the median session length of around 100 seconds. Videos with a medium popularity seem to have longer session times. CACTOS will utilize this difference to be able to improve the QoS while reducing wasted resources.

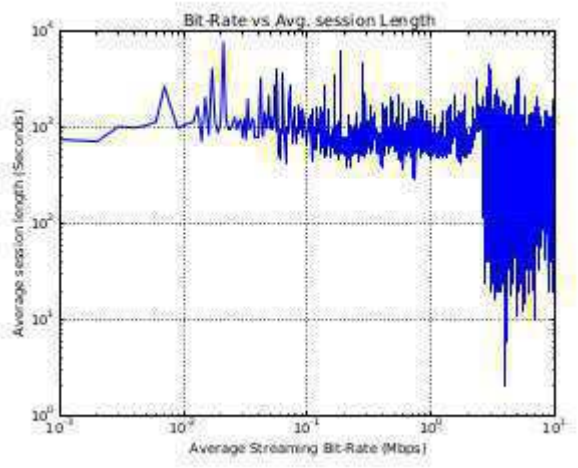
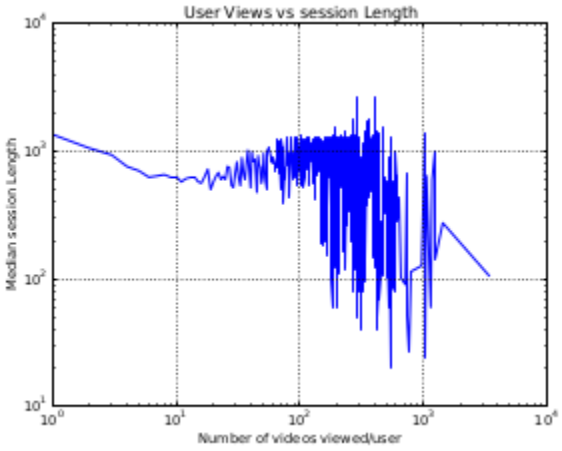


Figure 1819 Users impatience is not correlated with the bitrate of the video

To better understand why users abandon streams early, we investigated two hypotheses. The first hypothesis was that users abandon sessions due to low quality of the streaming, i.e., low bit-rate. Figure 18 shows how the average median session length of all users changes with the average streaming bit-rate. The figure shows that across most of the seen average bit-rates, the behaviour of impatient users does not change. Average bit-rates more than 3 Mbps are rare, and thus the variation seen when the bit-rates are more than 3 Mbps should not be interpreted as a change in the user-behaviour but rather as outliers. The second hypothesis was that users who use the service more will have a different average median session length. Figure 19 shows that the session length does not differ between users who use the service very often from those who do not. Thus, both hypotheses are not true. The session length distribution is an invariant in the system.



Figur 19 Heavy users of the service are generally more impatient than non-frequent users

THE MOLPRO APPLICATION

The Molpro application from the scientific computation scenario offers a set of chemical algorithms to calculate or simulate chemical computations. Therefore a scientist defines the requested computation in an input file for Molpro that has direct influence on the hardware affinity of the application behaviour. Since Molpro is normally executed on high performance computing (HPC) clusters, it requests two main tasks towards CactoOpt:

i. scheduling new incoming computations
ii. improving cluster utilisation and computations during runtime

Compared to the state of the art in HPC, the scheduling targets a more application aware placement of computations in a heterogeneous Cloud cluster. Improvements of hardware assignments during runtime are usually not available in common HPC clusters (cf. D7.1).

Application Behaviour

The application behaviour is directly influenced by two factors:

i. The specified input file which defines e.g. the chemical algorithm
ii. The used hardware for the computation and its utilisation by e.g. other applications

Depending on the input file different phases occur during the overall execution: Disk I/O bound or CPU bound. The phases come from a changing demand of hardware resources during the computation and lead to different hardware bottlenecks (Disk or CPU). Since we focus on single machine computations, networking is not considered but would be a third dimension.

The phases can be detected by looking at the ratio between CPU utilisation and disk I/O waiting time. Representing this ratio as a mathematical function offers possibilities like curve sketching to detect turning points in the ratio graph in order to define the beginning and ending of phases.

With the same input file, those phases occur predictably but with a different length depending on the used hardware. Changing the disk speed e.g. by replacing a HDD with a SSD leads to smaller I/O bound phases and hence to a shorter overall execution time (c.f. Table x). Multiple runs of the same computation on the same setup shows a very small deviation of the execution time with a SSD and a small deviation with a HDD[1].
Having a deeper look at the metrics offer the creation of more hardware independent metrics like the number of reads/writes to the disk or the number of CPU computations. Comparing this metrics from a same application input on different hardware confirms the similarity of application behaviour.

[1] Larger deviation with HDD caused by unpredictable placing of data on disk and, compared to SSD, very long seek times for reading.

Since the application behaves depending on the current bottleneck, additional applications on the same hardware at the same time with the same hardware demand will kill the overall execution time. Hence CactoOpt should be aware of that.

Table 1: Execution time and Deviation for Molpro with same Computation

Setup	*Average [min]*	*Deviation [min]*	*# Runs*
Intel i5 3.2GHz, 16GB Ram, SSD	385.14	0.36	3
Intel i5 3.2GHz, 16GB Ram, HDD	504.16	22.11	8

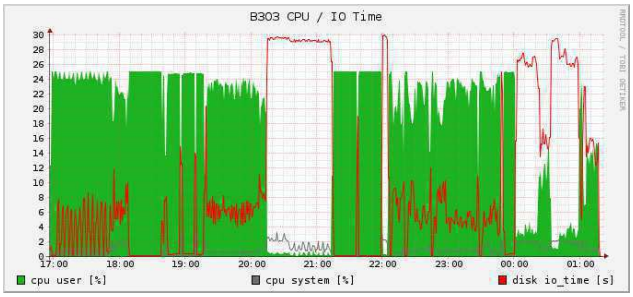


Figure 20 CPU/IO time as captured by Ganglia monitoring from UULM(HDD)

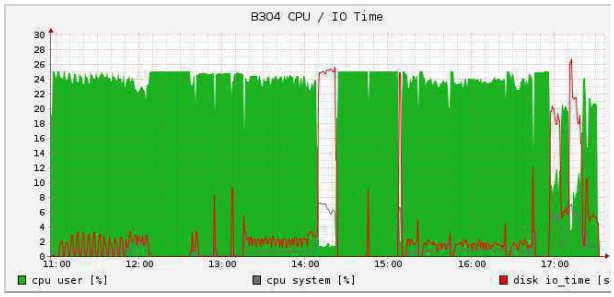


Figure 21 CPU/IO time as captured by Ganglia monitoring from UULM(SSD)

Setup	#1: CPU Bound	#2: I/O Bound	#3: CPU Bound	#4: Mixed
Intel i5 3.2GHz, 16GB Ram, HDD	94% CPU Usage 16.4M IOs 196m runtime	1% CPU Usage 40.9M IOs 50m runtime	89% CPU Usage 23.2M IOs 165m runtime	25% CPU Usage 96.3M IOs 80m runtime
Intel e5 2.6GHz, 16GB Ram, HDD	96% CPU Usage 14.3M IOs 219m runtime	2.4% CPU Usage 42.9M IOs 55m runtime	91% CPU Usage 20.9M IOs 185m runtime	31% CPU Usage 97.5M IOs 85m runtime

Inaccuracies caused by manual phase definition and rounding errors

Optimisations

Main optimisation goals when running Molpro applications in a Cloud setup are both from a Cloud user and Cloud provider viewpoint. The Cloud user, who submits the Molpro computation, requests a shorter execution time. The Cloud provider aims at perfectly utilise his hardware with respect to energy efficiency. Therefore non-conflicting Molpro computations could be placed on the same hardware node if the performance of the Users' computations are not touched but the overall cluster performance could be improved. Additionally the assignment of Molpro computations to available hardware can optimise the energy efficiency. The main optimisation factors summarised are:

i. Improve the (Cluster) Performance
ii. Improve the (Application) Execution
iii. Improve the (Overall) Energy Efficiency

To achieve those optimisation factors the actual phases of running applications and also upcoming phases in future can be considered. Therefore online mining can detect the current phase of running applications. The input file plus historical footprint checking of previous application runs can predict the next kind of phase. This output can then be used with a potential hardware to predict the execution time in future.

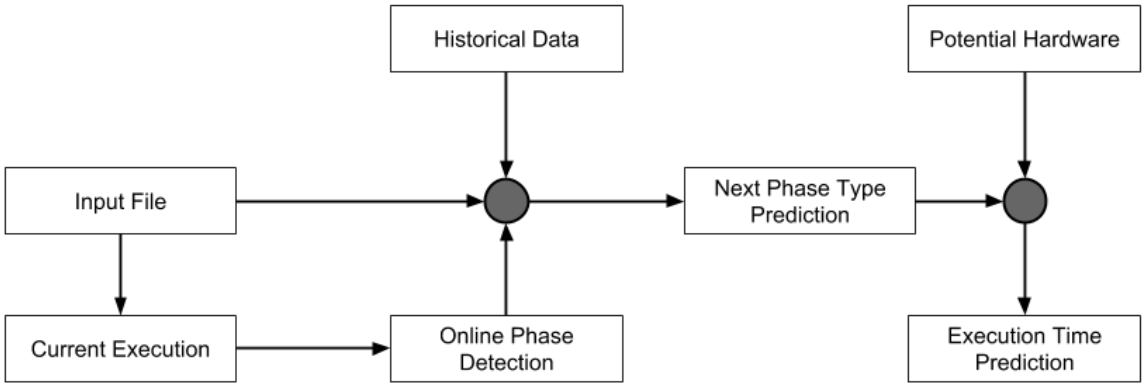


Figure 22 Steps to model and optimize the MolPro application.

I ELASTICITY MODELS AND PREDICTION TYPES

To illustrate the need for (and type of) prediction models used in this work, we here categorize and discuss cloud infrastructure optimization in the three categories of:

- Placement (allocation of virtual hosts to physical resources, including scheduling and migration).
- Vertical elasticity (dynamic control of virtual host capacity, e.g., CPU and RAM, assignments)
- Horizontal elasticity (dynamic control of instantiation of application components)

While all of these types of actions can benefit from load prediction, the type of information needed for the individual types of optimization well illustrates the CACTOS perspective of prediction modelling. Placement of virtual hosts is typically done in two phases - upon admission of new applications (and thus initial instantiation and bootstrapping of virtual hosts) as well as periodically (when monitoring of existing placements indicate opportunities for improvements). As such placement of virtual hosts needs (at least) coarse-grained estimates of the load that will be placed on physical resources in internal load dimensions (e.g., CPU, RAM, and I/O). In the CACTOS work we use the aforementioned component model upper and lower load bound estimations, as well as prediction models that model seasons and trends in component capacity requirements (i.e. based on application trace logs when available), for virtual host placement and scheduling.

Vertical elasticity techniques can be seen as feedback control based regulators of capacity, i.e. controllers that regulate what virtual host gets to use what capacity on a physical resource (i.e. what resources within a physical resource). Typically a set of virtual hosts are assigned to a physical resource and to enable these virtual hosts to be co-hosted without degraded application QoS, these controllers need accurate estimates of predicted component load. As vertical elasticity actuators typically are fast acting (i.e. with a short delay between control and action), the load prediction time window does not need to be very large for such a controller to be efficient. However, as vertical elasticity mechanisms by definition only control resource assignments for a single virtual host, vertical elasticity mechanisms have well defined limits for how much they can scale the capacity allocated to components.

Horizontal elasticity mechanisms can be seen as controllers that control instantiation of virtual hosts, e.g., load balancer control of duplication of virtual machines. As such, horizontal elasticity can scale out and thus provide greater increases in the capacity allocated to applications than vertical elasticity techniques. However, as instantiation of virtual hosts can be time consuming [13] (due to delays in, e.g., virtual machine instantiation and trigger (or require) migration of existing workloads, horizontal elasticity mechanisms typically operate using longer time perspectives and on application-level behaviour patterns (e.g., request arrival patterns). For the same reasons (actuator delays), it is also important to detect unexpected load peaks as early as possible to be able to scale up application resources.

Modelling of application and component behaviour is often done using time series analysis and signal processing techniques that operate on samplings of key performance indicator (KPI) values. As a number of user behaviour patterns (e.g., daily and weekly schedule patterns) often are evident at application level, user and application behaviour models are often based on analysis of application parameter data, e.g.,

application-level requests for web applications. For analysis of component load patterns, similar techniques can be used at component level, e.g., analysing resource capacity KPIs such as CPU and I/O patterns in component and physical resource log traces.

A Predictive Workload Model

As seen in Figure 3, the workload ($Y_{t)}$ has a pronounced repetitive weekly pattern, which slowly varies over time. We call this the (weekly) pattern, $P_{t,}$ an estimate it by fitting cubic splines to the data [22]. Cubic splines are flexible and able to pick up the features of the pattern. The workloads' deviation from the pattern ($res_t = Y_t - P_{t)}$ is called the residual and will typically have a positive auto correlated structure that also brings information about future values. It may be captured by an autoregressive model, e.g. in the simplest case with lag structure one,

$$res_{t+1} = a_0 + a_1 res_t + e_{t+1}, \quad (1)$$

where e_t is white noise. The workload is also characterized by occasional outliers, having extremely large or small values compared to the surrounding workload values. Outliers typically come in time consecutive groups. Downward outliers occur due to monitoring or system problems and do not reflect the true workload. Using bogus monitoring values for management decisions can lead to severe decrease in the application performance. They should therefore be ignored and the corresponding (unobservable) workload be predicted by the estimated pattern. Upward outliers (large values) on the other hand are real workload increases and should be predicted as accurately as possible. While it is not possible to predict a completely random event, i.e., foresee the first upward outlier in a group, it is important for the predictor to adapt quickly and start mitigating for the workload change. We thus predict the next workload value by the estimated pattern plus the prediction error of the last workload, thus aiming at catching up as fast as possible to the"explosive" nature of the upward outliers. If the last value was not an outlier, we predict the next workload by the estimated pattern plus the residual estimated from the autoregressive model in Equation 1.

The repetitive weekly pattern slowly changes with time in amplitude, level and shapes. Thus, when estimating the pattern and the residual AR model, and when identifying outliers, it has to be done locally, say using only the two last weeks of workload. Moreover the pattern and the AR model should be estimated without the influence of outliers. We propose to do the following: Let Y_t^* denote the true workload Y_t if it is not an outlier (to be defined), and let it correspond to the estimated pattern P_t otherwise. Suppose we have chosen a two week window of workload data without outliers, $Y^*_{t-335}, ..., Y^*_t$ and want to predict the workload at time t+1. Estimate the weekly pattern $Y_{t,}$ by first overlaying the two weeks of data on top of each other, such that there are two workloads for each hour over a week. Fit a cubic spline to these data, putting knots at every second hour of a week (87 knots in total over 168 hours of a week) and using B-splines as basis functions. Now compute the residuals,

$$res^*_h = Y^*_h - P_h, \; h = t - 335, ..., t,$$

noting that $P_h = P_h+168$, and estimate the coefficients of the AR model using Equation 1 by a least squares fit of

$$res^*_{t+1} = a_0 + a1\ res^*_t$$

If the last workload, Y_t, is not an outlier, predict the next workload, Y_{T+1}, by

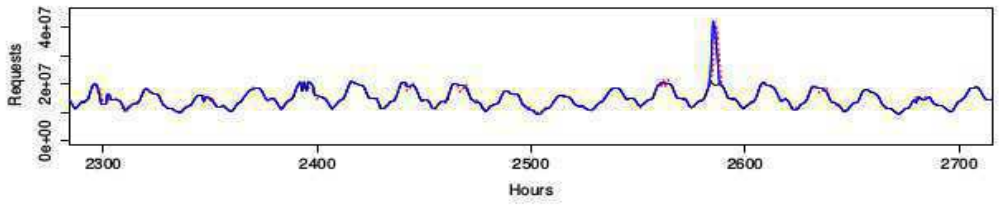
$$Y_{T+1} = P_{t+1} + a_0 + a_1 (Y_T - P_t).$$

If Y t is an upward outlier predict Y t+1, by

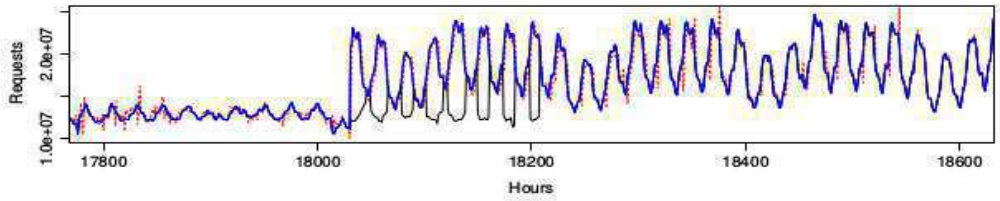
$$Y_{T+1} = P_{t+1} + (Y_T - P_t).$$

If Y t is a downward outlier predict Y t+1, by

$$Y_{T+1} = P_{t+1}.$$



(a) Predictions for the period between 21st of Aug. 2008 till the 6th of September, 2008.



(b) Predictions for the period between 29th of May 2010 till the 12th of Aug., 2010.

Figure 23 Using Cubic splines to predict the workload on Wikimedia servers

Outliers are defined as follows. First compute the standard deviation s_{res} of the residuals res^*_h, h = t−335,..., t. Suppose that we go forward in time and observe Y_{t+1}. Then Y_{t+1} is defined to be an upward outlier if Y_{t+1} is greater than the maximum value of the estimated pattern (over the last two weeks) plus $4s_{res}$. Moreover, Y_{t+1} is defined to be a downward outlier if it is zero or less than the minimum value of the estimated pattern (over the last two weeks) minus $4s_{res}$. To predict the workload at time t + 2, having observed Y_{t+1}, we slide the two week window one hour ahead in time and repeat the above algorithm. The predictive model was applied to both the flow of requests and the data flow for the Wikimedia workload.

Examples of prediction performance for the request flow can be seen in Figure 23. The actual workload is shown in blue solid lines, the predicted value in red dotted lines and the predicted value without compensation for the bursts in solid black lines. In Figure 23(a), there is a single short spike where the workload almost doubled, while in Figure 23(b), the workload almost doubled but stayed like that. It is clear that using our corrective mechanism, the predictor is able to rapidly cope to the changing workload dynamics. Comparable results were obtained for predicting the data flow that we omit due to lack of space. We calculated the Mean Absolute Percentage Error (MAPE) for our predictions for both the requests flows and the data flows. When excluding outliers, the MAPE for the predicted request flow is 2.2% while for the data flow is 2.3%, that is, on average we miss the true workload one step ahead by around 2% for both the data and request streams. When only considering spikes, the MAPE for the request stream is 4.6% and is 11% for the data stream.

II BURSTINESS MODELLING

Workload spikes and burstiness decrease online applications' performance and lead to reduced QoS and service disruptions [6]. We define a workload spike (sometimes referred to as a burst or a flash crowd[5]) to be a sudden increase in the demand on an object(s) hosted on an online server(s) due to an increase in the number of requests and/or a change in the request-type mix [29]. Some spikes occur due to a non-predictable event in time with non-predictable load volumes while others occur due to a planned event but with non-predictable load volumes.

A bursty workload is a workload having a significant number of spikes. The spikes make it harder to predict the future value of the load. Bursty workloads complicate cloud resource management since cloud providers host a multitude of applications with different workloads in their datacentres. Problems such as service admission control, Virtual Machine (VM) placement, VM migration and elasticity [17] are examples of resource management problems that are complicated due to workload spikes and burstiness. It is therefore important to be able to measure the burstiness of a workload in the CACTOS toolkit in order to be able to adapt the optimisation plans and perform resource management in an adaptive way. It is also interesting to be able to generate artificial workloads with different burstiness profiles that can be used by CactoSim to test what-if scenarios for deployment and optimization.

We identified some requirements for a burstiness metric to be robust and work on a wide range of scenarios.

1) The metric should be able to capture changes in a wide set of workload types.

2) The parameters used for calculating the metric should be intuitive, and therefore easy to set.

3) The metric should be able to operate on short data sequences and to be fast to compute.

4) The metric should differentiate between a gradual workload increase and a sudden one. For example, techniques using entropy are not able to do that.

Sample Entropy as a Burstiness Measure

Sample Entropy (SampEn) is a robust burstiness measure that was developed by Richman et al. over a decade ago [23]. It is used to classify abnormal (bursty) physiological signals. It was developed as an improvement to another burstiness measure, Approximate Entropy, widely used previously to characterize physiological signals [21]. Sample Entropy is defined as "the negative natural logarithm of the (empirical) conditional probability that sequences of length m similar point-wise within a tolerance r are also similar at the next point". It has two advantages over Shannon's entropy: i) being able to operate on short data sequences and, ii) it takes into account gradual workload increases and periodic bursts. These advantages make it an interesting potential measure for workload burstiness as a workload having periodic bursts, e.g., every weekend, is easier to manage compared to workloads with no repetitive bursts.

Three parameters are needed to calculate SampEn for a workload. The first parameter is the pattern length m, which is the size of the window in which the algorithm searches for repetitive bursty patterns. The second parameter is the deviation tolerance r which is the maximum increase in load between two

consecutive time units before considering this increase as a burst. The last parameter is the length of the workload which can easily be computed. We therefore focus on m and r and their choice. The deviation tolerance defines what a normal increase is and what a burst is. When choosing the deviation tolerance, the relative and absolute load variations should be taken in account, For example, a workload increase requiring 25 extra servers for a service having 1000 VMs running can probably be considered within normal operating limits, while if that increase was for a service having only 3 servers running then this is a significant burst. Thus by carefully choosing an adaptive r, SampEn becomes normalized for all workloads. If SampEn is equal to 0 then the workload has no bursts. The higher the value for SampEn, the more bursty the workload is.

Sample Entropy Implementation

There is one main limitation of SampEn, it is expensive to calculate both CPU-wise and memory-wise. The computational complexity (in both time and memory) of SampEn is O (n^2) where n is the number of points in the trace. In addition, workload characteristics might change during operation, e.g., when Michael Jackson died, 15% of all requests directed to Wikipedia were to the article about him creating spikes in the load. If SampEn is calculated for a long history, then recent changes are hidden by the history.

To address these two points, we modified the sample entropy algorithm by dividing the trace into smaller equal sub-traces. SampEn is calculated for each sub-trace. A weighted average, AvgSampEn, is then calculated for all SampEn values for the sub-traces. More weight can be given to more recent SampEn values. This way the time required for computing SampEn is reduced since n is reduced significantly. Our modification also enables online characterization of workloads since SampEn is not recomputed for the whole workload history but rather for the near past.

Our algorithm is shown in Algorithm 1. T is the workload for which SampEn is calculated. The trace is divided into N sub-traces of length L (lines 1 to 3). For each sub-trace, W, SampEn is calculated. The first loop in the algorithm (lines 9 to 14) calculates B^m (r), the estimate of the probability that two sequences in the workload having m measurements do not have bursts. The second loop in the algorithm (lines 15 to 19) calculates A^m (r), the estimate of the probability that two sequences in the workload having m + 1 measurements do not have bursts. Then SampEn for the sub-trace is calculated and is added to the sum of the SampEn values of all previous sub traces multiplied by a weighting factor a (line 20). The average SampEn for the whole trace is then calculated.

In order to not make the length of this report very long, we point the interested reader to our recently published paper where more details on the evaluation of Sample Entropy and how it compares to the State-of-the-Art in burstiness quantification is presented [3].

Data: r, m, T, L
Result: $AvgSampEn$

$N \leftarrow Length(L)$;
$P_{divided} \leftarrow \{T(L.k).....T(L.(k+1)\}\ \forall k \in \{0,N\}$;
$Tot_{SampEn} \leftarrow 0$;
for W *in* $P_{divided}$ **do**
  $n \leftarrow Length(W)$;
  $B_i \leftarrow 0$;
  $A_i \leftarrow 0$;
  $X_m \leftarrow \{X_m(i)|X_m(i) = [x(i),...x(i+m-1)]\ \forall\ 1 < i < n-m+1\}$;
  for $(X_m(i), X_m(j))$ *in* X_m: $i \neq j$ **do**
    Calculate $d[X_m(i), X_m(j)] = \max(|x(i+k) - x(j+k)|)\ \forall\ 0 \leq k < m$;
    if $d[X_m(i), X_m(j)] \leq r$ **then**
      $B_i \leftarrow B_i + 1$;
  $B^m(r) \leftarrow \frac{1}{n-m} \sum_{i=1}^{n-m} \frac{1}{n-m-1} B_i$;
  $m = m+1$
  $X_m \leftarrow \{X_m(i)|X_m(i) = [x(i),...x(i+m-1)]\ \forall\ 1 < i < n-m+1\}$;
  for $(X_m(i), X_m(j))$ *in* X_m: $i \neq j$ **do**
    Calculate $d[X_m(i), X_m(j)] = \max(|x(i+k) - x(j+k)|)\ \forall\ 0 \leq k < m$;
    if $d[X_m(i), X_m(j)] \leq r$ **then**
      $A_i \leftarrow A_i + 1$;
  $A^m(r) \leftarrow \frac{1}{n-m} \sum_{i=1}^{n-m} \frac{1}{n-m-1} A_i$;
  $Tot_{SampEn} \leftarrow |-\log[\frac{A^m(r)}{B^m(r)}]| + a \times Tot_{SampEn}$;
$AvgSampEn \leftarrow Tot_{SampEn}/N$;

Algorithm 1 Sample Entropy modified algorithm

III RELATED WORK

To further provide context to the work presented in this document, we here include a brief survey of workload modelling and resource management systems that are conceptually related to different aspects of the CACTOS approach to infrastructure optimization and modelling.

MISTRAL: DYNAMICALLY MANAGING POWER, PERFORMANCE, AND ADAPTATION COST IN CLOUD INFRASTRUCTURES

Goal: Mistral [15] is a holistic controller framework that optimizes trade-offs among power consumption, application performance, and adaptation costs.

Application performance objective is specified in terms of a target mean response time. To include performance into optimization formula actual response time is compared with target one and reward for meeting it or penalty for missing it is applied.

In order to calculate the total cost of applying an adaptation Mistral considers: adaptation duration, increased response time of applications during applying adaptation (both for involved and co-located applications), and increased power consumption during applying adaptation.

Types of actions considered: Controller uses following adaptation actions to improve the data center configuration: increase/decrease VM's CPU capacity, add/remove VM, migrate VM, shut down/restart physical machine.

Algorithm: Mistral controls the costs of search versus the potential benefits during generation of adaptation decisions. It considers its own power consumption and reduces the search space by using a heuristic to estimate adaptations costs and comparing the intermediate solutions with the ideal configuration.

To handle large-scale infrastructures multi-level hierarchy of controllers is introduced, where lower-level controllers manage small number of physical machines at finer time granularity, while higher-level controllers coordinate work of lower-level controllers at coarse grained time granularity.

SLA-BASED OPTIMIZATION OF POWER AND MIGRATION COST IN CLOUD COMPUTING

Goal: The objective is to minimize the total operational cost of the system including power and migration costs, and penalties for violating response time constraints [11].
Power consumption is modelled as a sum of constant consumption for idle machine and variable part related to the utilization of the server.

Types of actions considered: Types of optimization actions considered: switching physical machines on/off, migrating VM, vertical scaling of VM.

Algorithm: To find the best configuration of data center a heuristic algorithm based on convex optimization and dynamic programming is used.

PMAPPER: POWER AND MIGRATION COST AWARE APPLICATION PLACEMENT IN VIRTUALIZED SYSTEMS

Goal: pMapper is an application placement controller that dynamically places applications to minimize power while meeting performance guarantees. Hence, performance is not a metric to be maximized, but a constraint that has to be fulfilled [30].

Migration is characterized by a migration duration and a migration cost, where cost is revenue loss because of the decreased performance of applications during migration estimated by quantifying the decrease in throughput.

Types of actions considered: Controller is designed to utilize following power management actions: CPU idling in the hypervisor, DVFS and throttling, and VM migration.

Algorithm: min Power Parity (mPP) algorithm works in two phases: firstly, it determines a target utilization for each server based on the power model, and secondly, it places VMs on the servers using incremental First Fit Decreasing (*iFFD*). *iFFD* first identifies servers with current utilization different from the target one and divides them into two groups: receivers that are over utilized, and donors, that are underutilized. Then for each donor it selects the smallest applications to migrate and stores them on a VM migration list. Finally it decides where to migrate VMs using FFD with the spare capacity on the receivers as the bin size and the VM migration list as the balls.

OMEGA: FLEXIBLE, SCALABLE SCHEDULERS FOR LARGE COMPUTE CLUSTERS

Goal: Omega [25] is a cluster management system that coordinates parallel schedulers. It is built around shared state concept and uses lock-free optimistic concurrency control. Its main goal is to achieve both implementation extensibility and performance scalability.

Types of actions considered: The system considers only initial placement decisions – once the tasks are placed they are not scaled or migrated.

Algorithm: There is no central resource allocator in Omega, however to coordinate schedulers, a master copy of resource allocation is maintained. To increase parallelization, each scheduler has its own local and frequently-updated copy of resource allocation. Once a scheduler makes a placement decision, it updates the master copy. If necessary, conflicts are resolved and local copies of resource allocation are resynced.

AGILE: ELASTIC DISTRIBUTED RESOURCE SCALING FOR INFRASTRUCTURE-AS-A-SERVICE.

Goal: AGILE [19] dynamically and proactively adjusts the number of VM assigned to a cloud application in a way that minimizes the costs of infrastructure provisioning and penalties imposed due to SLO violations.

Types of actions considered: AGILE uses pre-copy live cloning to replicate running VM to achieve immediate performance scale up.

Algorithm: AGILE predicts a future workload using a wavelet-based resource demand model, which looks ahead for up to 2 minutes – the time needed to clone a VM. Then, it uses an application-agnostic resource pressure model to map the application's SLO violation rate target into a resource pressure – the ration of resource usage to allocation.

Nguyen, Hiep, et al. "Agile: Elastic distributed resource scaling for infrastructure-as-a-service." Proc. of the USENIX International Conference on Automated Computing (ICAC'13). San Jose, CA. 2013.

IV REFERENCES

[1] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212, April 2012.

[2] Ahmed Ali-Eldin, Maria Kihl, Johan Tordsson, and Erik Elmroth. Analysis and characterization of a video-on-demand service workload. In *ACM MMSys*, page to appear. ACM, 2015.

[3] Ahmed Ali-Eldin, Oleg Seleznjev, Sara Sjostedt-de Luna, Johan Tordsson, and Erik Elmroth. Measuring cloud workload burstiness. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 566–572. IEEE, 2014.

[4] Ahmed Ali-Eldin, Johan Tordsson, Erik Elmroth, and Maria Kihl. Workload classification for efficient autoscaling of cloud resources. Technical report, Technical Report, 2005.[Online]. Available: h ttp://www8. cs. umu. se/research/uminf/reports/2013/013/part1. pdf, 2013.

[5] Ismail Ari, Bo Hong, Ethan L Miller, Scott A Brandt, and Darrell DE Long. Managing flash crowds on the internet. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*, pages 246–249. IEEE, 2003.

[6] Peter Bodik, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 241–252. ACM, 2010.

[7] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[8] Adrian Cockcroft. Cloud native cost optimization, 2014.

[9] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[10] Ahmed Ali Eldin, Ali Rezaie, Amardeep Mehta, Stanislav Razroev, Sara Sjostedt de Luna, Oleg Seleznjev, Johan Tordsson, and Erik Elmroth. How will your workload look like in 6 years? analyzing wikimedia's workload. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 349–354. IEEE, 2014.

[11] Hadi Goudarzi, Mohammad Ghasemazar, and Massoud Pedram. Sla-based optimization of power and migration cost in cloud computing. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 172–179. IEEE, 2012.

[12] Norden E Huang, Zheng Shen, Steven R Long, Manli C Wu, Hsing H Shih, Quanan Zheng, Nai-Chyuan Yen, Chi Chao Tung, and Henry H Liu. The empirical mode decomposition and the hilbert spectrum for nonlinear and non-stationary time series analysis. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 454, pages 903–995. The Royal Society, 1998.

[13] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pages 104–113. IEEE, 2011.

[14] Shudong Jin and Azer Bestavros. G ismo: a generator of internet streaming media objects and workloads. *ACM SIGMETRICS Performance Evaluation Review*, 29(3):2–10, 2001.

[15] Gueyoung Jung, Matti A Hiltunen, Kaustubh R Joshi, Richard D Schlichting, and Calton Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 62–73. IEEE, 2010.

[16] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 117–126, New York, NY, USA, 2009. ACM.

[17] Maria Kihl, Erik Elmroth, Johan Tordsson, Karl-Erik Årzén, and Anders Robertsson. The challenge of cloud control. In *8th International Workshop on Feedback Computing*, 2013.

[18] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711. ACM, 2014.

[19] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the USENIX International Conference on Automated Computing (ICAC'13). San Jose, CA*, 2013.

[20] Vern Paxson and Sally Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking (ToN)*, 3(3):226–244, 1995.

[21] Steven M Pincus. Approximate entropy as a measure of system complexity. *Proceedings of the National Academy of Sciences*, 88(6):2297–2301, 1991.

[22] James O Ramsay. *Functional data analysis*. Wiley Online Library, 2006.

[23] Joshua S Richman and J Randall Moorman. Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6):H2039–H2049, 2000.

[24] Gabriel Rilling, Patrick Flandrin, Paulo Goncalves, et al. On empirical mode decomposition and its algorithms. In *IEEE-EURASIP workshop on nonlinear signal and image processing*, volume 3, pages 8–11. NSIP-03, Grado (I), 2003.

[25] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[26] L. Tomas and J. Tordsson. An autonomic approach to risk-aware data center overbooking. *Cloud Computing, IEEE Transactions on*, 2(3):292–305, July 2014.

[27] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[28] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review*, 36(SI):239–254, 2002.

[29] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[30] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *Middleware 2008*, pages 243–264. Springer, 2008.

[31] Hans-Joachim Werner, Peter J. Knowles, Gerald Knizia, Frederick R. Manby, and Martin SchÃ¼tz. Molpro: a general-purpose quantum chemistry program package. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 2(2):242–253, 2012.

[32] Hongliang Yu, Dongdong Zheng, Ben Y Zhao, and Weimin Zheng. Understanding user behavior in large-scale video-on-demand systems. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 333–344. ACM, 2006.

V APPENDIX A: WORKLOADS

For reference, we here include more thorough information about the workloads used and why they have been selected for application behaviour modelling in this work.

THE VOD WORKLOAD

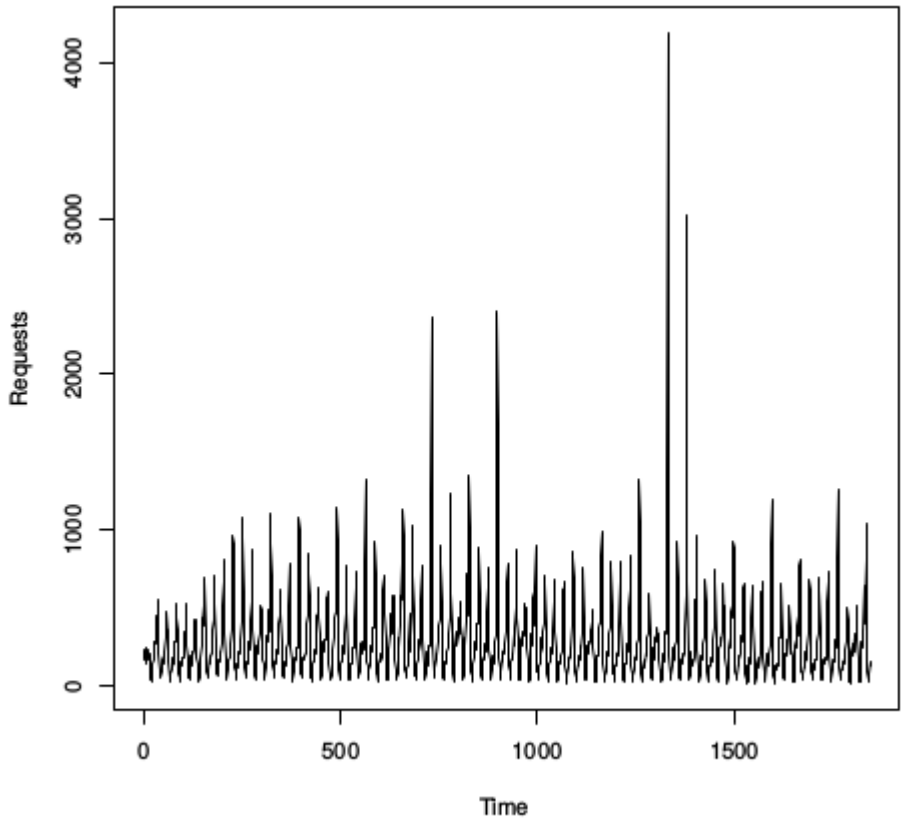


Figure 24 The VoD workload request pattern

Over the past decade, Video on Demand (VoD) and Video sharing online services have been on the rise. A recent report estimated that more than 50% of the total downstream traffic during peak periods in North America originate from Netflix and YouTube. It is thus required to analyze and characterize VoD workloads in order to understand how to improve and optimize the network usage and the perceived Quality-of-Service (QoS) by the service users. Many VoD service providers utilize the power of cloud computing to host their services. Since a typical cloud hosts multitudes of applications with differing workload profiles.

Cloud service providers need to understand the workload characteristics of the running applications including the VoD workload dynamics. This understanding is crucial as application co-hosting can result in performance interference between collocated workloads. To better understand VoD workloads, we

obtained recent workload traces from TV4, a major Swedish VoD service provider, detailing the requests issued by the premium service subscribers to TV4's VoD service. The VoD service is hosted on a number of cloud platforms.

The traces contain logged data between December 31 2012 and March 18 2013 from two cities with a total population over half a million. The traces therefore represent a typical medium European city. Figure shows the video request arrival rate with time.

THE WIKIMEDIA FOUNDATION WORKLOAD

We analyze the workload of a large-scale website, representing a typical application for the cloud. The selected workload is from the Wikimedia foundation servers, mostly known for operating Wikipedia, the sixth most popular site on the web. While three months of this workload has been analyzed previously [27], we analyze a much larger data set spanning the period between June 2008 and October 2013, making this one of the largest workload studies we are aware of.

At the beginning of our study, the foundation was operating all its projects including Wikipedia using around 300 Servers in Florida, which acted as the primary site, 26 in Amsterdam and 23 in Korea as shown in Figure 25. In January, 2013, the foundation was running around 885 servers and building a new cluster in San Francisco that went in production in March, 2013. Today, the foundation's servers are distributed on 5 different sites; Virginia, which acts as a primary site, Florida and San Francisco in the United States and two cluster in Amsterdam, the Netherlands.

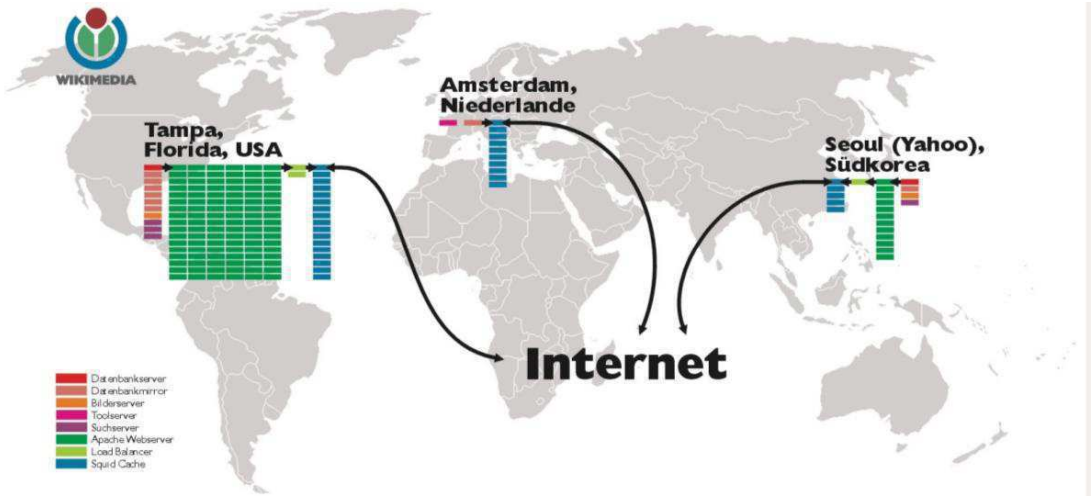


Figure 25 Wikipedia datacenters

The dataset studied consists of hourly logged files where each file contains the total number of requests directed to a page hosted on the Wikimedia foundation's servers, the page's name, to which project it

belongs and the total amount of bytes transferred for the page. The files sizes are between 20 MB and 120 MB of compressed data. While the logging started in December 2007