Bringing Scalability/Failover to a complex producer/consumer implementation

J. Houtman

A master thesis submitted to the

University of Twente, Enschede, the Netherlands

Department of Electrical Engineering, Mathematics and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science

Commissioned by: Startphone limited (workname Hyves)

August 2009

Graduation Comittee:
Ir. Pierre Jansen
Ir. Hans Scholten
Ir. Philip Hölzenspies
Drs. Reinoud Elhorst

Contents

Preamble

Popular dynamic websites employ a wide diversity of techniques to improve performance. Classic examples are database replication [[4], [3]], load balancing [11], caching [[11],[39]], optimizing web pages [32], the application of the producer/consumer[31] paradigm to offload heavy tasks from the website front-end[1], and data preparation for easy access by the website front-end.

In one way or another, all of these techniques are employed to increase the website performance of `http://www.hyves.nl`, the most popular social networking website in the Netherlands[2]. The latter two examples in the enumeration, offloading heavy tasks and data preparation, were adopted early on and have proven their value in various tasks[3], from sending e-mail notifications to importing blogs and photos from other on-line services. Hyves implemented these techniques in a statically configured system that had been designed when the website was a lot smaller. This resulted in a system that is scalable in terms of throughput, but requires too much maintenance due to its static configuration.

This research explores new solutions for implementing the mentioned techniques in a system with more flexibility to address current and future issues. It aims to improve the manageability of this system during its expected growth over the next few years.

[1] Servers communicating directly with the users

[2] `http://www.yme.nl/ymerce/2008/03/16/social-networking-diensten-in-nederland/`

[3] there are no hard numbers on this, but applying these techniques is an important corner stone in optimizing website performance

Preface

The last eighteen months have been spent completing this master's thesis in partial fulfillment of the requirements for the degree of Master of Science at the University of Twente. It is in fact a fine completion of a much longer, 6 year, period. As a novice I did my bachelor thesis at Hyves. The period since then has been spent at the university and partially at Hyves. Doing a second thesis project at Hyves gave obvious possibilities for a comparison between the two projects and nicely illustrated reached goals over the past few years both on a personal and professional level, this was an added bonus. I would like to thank a number of people. In order of appearance: R. Elhorst for taking on an inexperienced student 5 years ago; the University of Twente for offering a challenging educational program with competent teachers; P.G. Jansen, H. Scholten and P. Hölzenspies for providing supervision and guidance from an educational point of view; R. Elhorst (again) for the supervision, advice, guidance and criticism which he offered in his role of internal advisor; my direct colleagues for advice during this project and proofreading the thesis; and last but certainly not least my wife, who was a hawk when it came to correcting spelling and made it possible for me to concentrate on finishing this thesis during this rather dynamic period of our lives.

Summary

This research centers around two subjects. First queues used in a producer/consumer paradigm and implemented using a database must be made self-scalable. Secondly a self-scalable system for the queue consumers and h-workers (programs that perform tasks like data preparation) must be developed. Two solutions were suggested. The first solution creates a container from a fixed set of processors or h-workers, when required a queue is also added. A container is then run on a node, effectively creating a consumer side queue. Containers only hold h-workers or processors that perform the same task, to scale capacity for a task more containers are created. The de-centralized nature of the solution only allows for gradual scaling. This is a very ill fit for the h-workers, because they run in intervals and have a very abrupt need for resources.

The second solution separates the queues from the consumers on a conceptual level. This allows different systems for the queues and the consumers/h-workers. Each queue is segmented into a minimum of two segments and divided over a set of nodes. A weight associated with each queue segment determines how the incoming events are distributed over the available segments.

Queues are typed and different types can coexist on the same node. Incoming events are routed to appropriately typed queue-segments. Scaling takes place by changing the routing of events to be better spread over the available nodes. Consumers and h-workers are implemented in a master/worker model (further called manager/worker) which runs on top of a batch system, thereby allowing the manager to request the necessary resources on demand. As long as there are enough resources avail-able these requests are met. This results in a system in which both continuous and abrupt resource demands can be met.

Several reasons, including the lack of performance data per queue (consumers) or task (h-workers) and ill fit scaling possibilities of the first solution, make solution two stand out. It is subsequently developed into a proof of concept. Data retrieved from the proof of concept indicates a partial success. The queue system is in its current state unusable, mostly due to the lack of performance statistics per queue segment. It is therefore unknown what weight should be redistributed, making it a guess game. The manager/worker implementation works well for consumers. The setup is able to adjust its resource consumption to the demands while timely processing all incoming events. Due to facilities provided by the batch system, the manager/worker paradigm is robust and quite resistant to failure. The h-workers have not been tested, but provided there are enough resources available in the batch system a manager can request the required resources and receive them after a small delay.

Samenvatting

Dit onderzoek beslaat twee onderwerpen. Ten eerste moeten, door middel van een database, gemplementeerde queues die gebruikt worden in een producer/consumer paradigm, automatisch schaalbaar gemaakt worden. Ten tweede moet een automatisch schaalbaar systeem ontwikkeld worden voor de queue consumers en de h-workers (programmas die taken vervullen zoals data voorbereiding). Er zijn twee voorgestelde oplossingen. In de eerste oplossing wordt een container gecreerd die bestaat uit een vaste set processors of h-workers. Indien nodig wordt ook een queue toegevoegd, waarmee een consumer side queue ontstaat. Deze container draait dan op een node. Alle h-workers of processors in een container vervullen dezelfde taak. Om de capaciteit voor deze taak uit te breiden worden meer containers ingezet. De decentrale opzet van deze oplossing faciliteert alleen geleidelijke verandering in capaciteit. Dit maakt deze methode zeer ongeschikt voor h-workers, omdat die met tussenpozen draaien en sterk schommelende capaciteitsbehoeften hebben.

De tweede oplossing scheidt op een conceptueel niveau de queues van de consumers. Dit biedt de mogelijkheid van verschillende systemen; n voor de queues en n voor de consumers/h-workers. Elke queue wordt opgesplitst in minstens twee segmenten die worden verdeeld over een set nodes. Elk queue-segment krijgt een gewicht toebedeeld, aan de hand waarvan de binnenkomende events verdeeld worden over de beschikbare segmenten. Queues worden ingedeeld in types en verschillende types kunnen samen op dezelfde node draaien. Binnenkomende events worden gerouteerd naar het juiste type queue-segment. De capaciteit wordt aangepast door het gewicht te herverdelen over de beschikbare nodes. Consumers en h-workers worden geimplementeerd in een master/worker model (hierna manager/worker genoemd), dat bovenop een batchsysteem draait. Dit stelt de manager in staat op elk moment om meer capaciteit te vragen. Zo lang de capaciteit het toestaat, wordt aan deze vraag voldaan. In dit systeem wordt in zowel de langlopende als in de plotselinge behoefte aan capaciteit voorzien.

Verschillende punten, waaronder het gebrek aan performance-data per queue (consumers) of taak (h-workers) en de moeizame schaalbaarheid van de eerste oplossing, maken dat de voorkeur uitgaat naar de tweede. Deze wordt ontwikkeld tot een proof of concept. De data die hieruit voortkomen, wijzen op een gedeeltelijk succes. Het queue systeem is in zijn huidige staat niet bruikbaar. Dit is voornamelijk te wijten aan het gebrek aan performance statistics per queue-segment. Door dit gebrek is niet bekend welk gewicht herverdeeld moet worden, wat resulteert in giswerk. De manager/worker implementatie echter, werkt goed voor consumers. Deze constructie is in staat haar capaciteitsgebruik aan te passen aan de vraag en alle binnenkomende

events tijdig te verwerken. Door de eigenschappen van het batchsysteem, is het manager/worker paradigm robuust en behoorlijk foutbestendig. De h-workers zijn niet getest, maar uitgaand van voldoende capaciteit in het batch system kan een manager de vereiste capaciteit opvragen en er binnen korte tijd over beschikken.

Chapter 1

Introduction

Since the rise of the internet, especially the last five years, many services have emerged to connect friends through the internet. Well known examples are msn[21] and, in the category of social networking services, MySpace[26], Facebook[8] and the Dutch website Hyves[13].

An on-line social network service aims to build on-line communities of people who share interests and/or activities. It creates a place for people to interact with each other using a variety of services and media. Users mostly interact through messages but other media like photo's, music, videos and specially written applications like the open social gadgets are popular as well. While the majority of user interaction takes place through a website, other media like e-mail, instant messaging and SMS are also integrated into popular social networking services.

The formentioned website `http://www.hyves.nl` is, in the Netherlands, by far the biggest in its kind and is akin to popular international sites like myspace[26] and hi5[12]. Beside this Hyves.nl is the third most popular dutch website, following the dutch versions of google[9] and windows live[18]. It is also number 165 in the world ranking[1].

These rankings bring interesting opportunities and challenges in all aspects of the site, such as:

- The possibility to use this dominant position in the Netherlands to generate revenue, or even profit.
- The development of features that take advantage of the high percentage of the youth present on Hyves.
- Handling privacy considerations.
- Tackling copyright issues for the content uploaded by users.
- Serving 5 billion page views each month in a timely fashion.
- Improving the manageability of the almost 2000 servers (also called nodes and machines).
- Serving and storing more than 400 Tbytes of photos and music.

[1] according to `http://www.alexa.com` on 29 Nov 2008

- Managing and prioritizing the infinite number of items on the todo list.
- Efficiently handling and foreseeing scalability issues.

Even though Hyves is the third largest website in the Netherlands a steady expansion is still measured and considerable growth in both the amount of users and the number of photo's/message/etc per user is expected[2].

1.1 Hyves Architecture

Even though this thesis effects only a small portion of the underlying system architecture at Hyves a broad overview is given here to create an insight into the complexity involved and to introduce and explain common terms (used in this thesis).

The Hyves serverpark can be divided into a front and back-end segment, this abstraction serves to make a general division between the servers that interact "directly" with the user (front-end) and the servers that support the front-end but have no direct connection with the user (back-end). By servers that interact "directly" with the user, we mean the servers that process requests coming directly from the users computer. As can be seen in figure figure 1.1 on the facing page, the front-end and back-end are divided into clusters. Each of these clusters is a group of servers performing a specific (set of) function(s). These clusters are formed for a variety of reasons; incompatible designs, specific optimizations or simply a dedication of hardware for performance reasons. Each cluster is named according to its function: Web cluster, Media cluster, etc.

1.1.1 Front end

The front-end consists of all clusters that communicate directly with the users.

The most prominent of these is the web cluster, the webservers contain all interface and business logic that require user interaction. The webservers handle the user requests and after retrieving the required data from the back-end, compile the resulting page and send it to the users browser (client). The clients web browser then renders the page and requests all external resources (images, layout specifications).

By reducing the response times of webservers and by minimizing the browser's render and load times, the user experience is optimized. This is achieved by extensive use of asynchronous loading of page content "just-in-time"[3]. This prevents unnecessary communication and rerenders pages only partially, reducing the load on the front-end and preventing unnecessary content fetching and generation on the backend.

Browser load times are further reduced by increasing the number of parallel connections the browser uses to fetch external resources. This makes more effective use of now widely spread high bandwidth internet connections. In general it can be said that a characteristic of all code in the front-end is, that it is designed to minimize the response time for the users.

[2]This expectation is a given in this thesis, as it is implied in the assignment.

[3]This is done using Asynchronous JavaScript[37] and XML (acronym: AJAX)

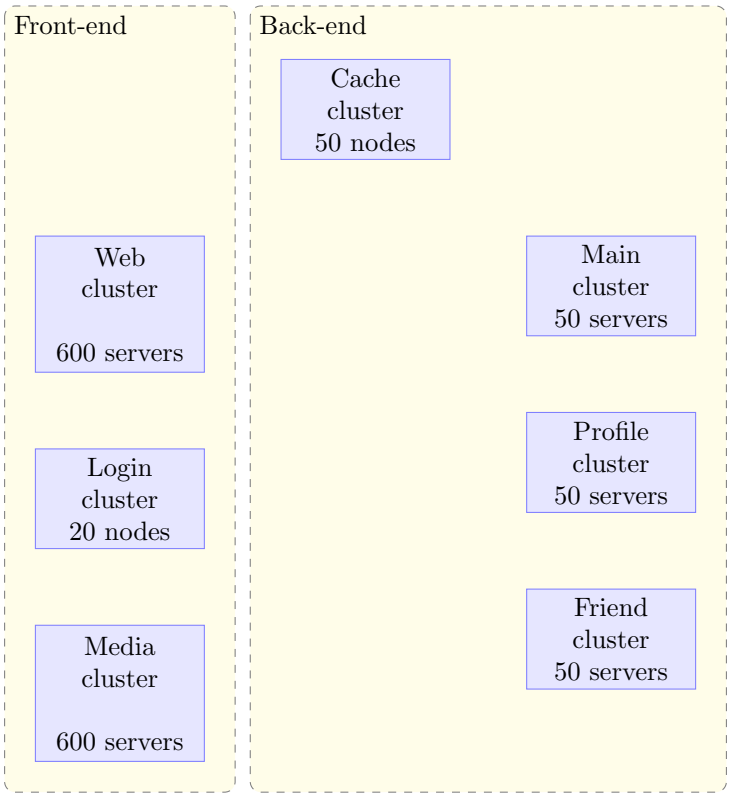

Figure 1.1: logical division of the hyves serverpark

Another large group of servers in the front-end is the media cluster . This cluster of over 600 servers handles 30.000 requests/sec and stores 350 million media items (audio, images and video) taking up about 1 petabyte of storage space. This architecture is kept scalable and fault-tolerant. Each media item is stored on two distinct servers in the cluster. This primary and secondary location of each media-item is kept in a large index. Each media item is served from its primary location, unless a server fails (its inability to perform its intended function)upon which it is served from the secondary location. Upon failure of a server the data on that server is considered lost and all media-items on that server have only one location left from which they can be served. A process is started which copies the media-items from that one location to a different server and the index is updated to reflect the new primary or secondary location of each media-item.

1.1.2 Back end

The back-end is considered to be everything else and performs two basic functions: Data storage/retrieval and the processing of all tasks that do not need user interaction.

Because the data-set used by Hyves is very large, data storage is distributed over multiple database clusters, as shown in figure 1.1. The data-set is split

up into disjointed subsets. Each of these subsets is stored in its own database. The subsets are chosen in such a way, that (closely) related information resides in the same subset, e.g. the caption of a photo should be in the same subset as that photo and the other photos within the same album should preferably also reside in that subset.

The data in our databases is most often stored in a normalized[6] way, this eases data manipulation and reduces storage space considerably. Due to its nature this method is not one of the fastest when data is retrieved, because the data needs to be searched, gathered and combined before it can be returned to the client.

To improve performance an application level cache is built between the database clusters and the front-end. Upon retrieval of cache-ble data from the database the front-end will store the (de-normalized) data in the cache, from where it can be retrieved the next time the front-end needs that data. Invalidation of the cached data can be done explicitly by the front-end when the data is updated or after a period of time, depending on the consistency restraints for that data. The cache allows for simple key-value storage and retrieval of data stored in memory, which leads to the faster data retrieval times.

The data storage/retrieval is comprised of various database architectures that provide permanent storage of the data at the cost of slower data retrieval. The cache provides fast data retrieval at the cost of storage efficiency and non-permanent data storage.

This leaves the second function of the back-end, the tasks that do not need user interaction. A task is a definite piece of functionality, for example the delivery of messages between Hyves users. This set of tasks is varied but can be separated into three categories, all of which are aimed at improving the response time of the website.

The first of the three categories is pre-fetching tasks. Pre-fetching is the process of fetching and storing data that is located externally to Hyves in our own back-end, this considerably improves data retrieval time. All pre-fetching tasks are executed regularly in order to keep the data from becoming stale.

Data preparation is the second category of tasks. Data preparation is the processing of our own data into a more efficient format that reduces retrieval time. For example, raw statistical data is processed throughout the day and then re-inserted into the back-end after which the results can be retrieved by the front-end.

Pre-fetching and data-preparation both reduce data-retrieval time at the cost of retrieving or preparing data that is never requested.

The third and final category of tasks are offloading tasks. These tasks execute a resource intensive function, in the background, that does not need user-interaction but does require parameters specified by the user. The function thus has no influence on the page generation time. The delivering of a message is such a task. After the front-end has gathered all needed parameters for the task (such as: title, body and recipients) it will send the parameters to the back-end where the actual delivery of the message takes place, this is called offloading.

For managing the execution of all these tasks a system called hyves-daemons has been setup and expanded over the years. The hyves-deamons system is a setup/deploy system. For each task it takes a number of arguments including

the program to run, a list of servers on which to run the program and how many instances of the program to run on each of those servers. On deploy the system sets up the tasks to run on the servers and starts the programs. This allows each task to exploit the data parallelism of the accessed data to the fullest.

The simplicity of the system allows it to scale to a large number of nodes and tasks because once deployed, no overhead and bottlenecks exists. The scalability of a tasks however can be limited, and is often the result of bottlenecks on data access.

1.2 Problem statement

The set of tasks in the hyves-daemon system and the servers in the cluster have grown and while the hyves-daemon system itself still has no bottlenecks, the popularity of the system has shown deficiencies in the design that need to be fixed. While these issues are strictly speaking not scalability issues their impact is closely connected to the popularity/size of the system.

Core to the noted deficiencies are the following disadvantages:

- The number of servers and instances per task are statically configured
- Performance data per task or instance is unavailable.

The deficiencies leading from this are:

- The lack of performance data per task or instance makes it difficult to analyze variety of situations, including over-utilization of a server, capacity requirements of a task and analyzing the performance of tasks.
- The static configuration hampers functionality like automatic load-balancing of the instances over the available servers and fail-over of instances to other servers upon failure of a server.
- Other inefficiency problems arise because all tasks are configured for the peak-load needed, this wastes resources during off-hours.
- All these currently manual operations require a constant stream of attention from the operators which would be well spent on other tasks. By improving on the points above would improve the maintainability of the system as it grows and thus also its scalability.

The problem statement can now be formalized as:

"How can the scalability, fail over and load balancing characteristics of the offloaded and data preparation tasks be improved, while decreasing (or at least not increasing) the workload for the technology department?"

While the problem statement is intended to give direction and focus to this research, it is not enough to explain its starting position and intended focus. These factors can, and will, be explained by exploring the state of art in chapter 2 on page 7 and the focus points defined below.

1.3 Research focus

In order to facilitate consistency in the decisions made, especially when there are conflicting interests, a prioritized list of focus points has been defined. Listed according to importance (Descending):

- Scalability of each task. As a guideline: a task should be able to grow tenfold without problems
- Failure resistence/ self healing. The system must have the ability to deal with failures, most importantly the failure of a server.
- No (major) modification of existing code. This research is not intended to redesign the whole system but should try to build on the code already in place.
- Support for single instance tasks. Tasks that can only run one instance at a time due to data-corruption issues should be supported.
- Automation. Common tasks in the system should be automated, for example resolving over-utilization of a server.
- Monitoring. Better support for (performance) monitoring

Of limited importance are:

- Efficient use of hardware. This is closely connected to scalability, but a scalable solution might still make in-efficient use of its hardware.
- Prioritization (at overload). When the required resources for all tasks exceed the capacity of the available servers, prioritization between tasks should be applied.
- Software/hardware prerequisites per task. Whenever possible the designed system should take into account prerequisites of tasks.

Chapter 2

State of Art

This section will describe the state of art at Hyves and existing techniques on the market. It will discuss benefits of these systems and the problems/limitations that are inherent to the techniques that are used. This will of course, happen in the context of this research and the problems it intends to solve.

The Hyves website uses two designs to implement the three categories of tasks that where discussed in section 1.2 on page 5. The first design is explained in section 2.1 and discusses the first two categories, pre-fetching and data preparation tasks. The second design is explained in section 2.2 on page 11 and covers the third category: offloaded tasks. The sections discuss the theory, implementation, advantages and weaknesses of each design.

The last two sections of this chapter explore virtual machines and batch systems and discusses their weaker and stronger points in order to decide on their suitability as a solution.

2.1 Implementation of pre-fetching data preparation tasks

Pre-fetching and data preparation tasks have been introduced because performing these tasks in the front-end significantly increases the response time of the website. The task are performed in the back-end and the result inserted to our own databases so that the front-end can retrieve the prepared data quickly when requested. This allows for the data to become stale, but this is remedied by running the tasks at regular intervals and when needed the task runs perpetually.

2.1.1 Tasks

A small example list of the tasks that are implemented this way:

- Photo email importer (pre-fetching)
- Server management data import (pre-fetching)
- Member integrity checker (data preparation)

The full list of tasks implemented using the h-worker principle can be seen in section A.2 on page 58

2.1.2 Technique

The programs implementing tasks of these types are called h-workers. A h-worker will, on execution perform the whole, or more commonly, a small subset of the task it is designed to perform. Each execution of the program is said to create an instance of that program, multiple instances can be started so that they run in parallel. Either parallel or sequential, it is the set of these instances that perform the whole task.

Upon execution a h-worker retrieves the list of work items through some arbitrary method and starts processing. After finishing this work the instance quits. When an instance only performs a small subset of the total task, it will only retrieve a small set of the work that needs to be done, restarting the h-worker instance after it quits ensures that the whole task gets done eventually.

When the data accessed by a h-worker is suitable for parallel access, multiple instances of an h-worker can run simultaneously to decrease the total runtime of a task. If not, for example because parallel access will lead to data-integrity issues, only one instance of that h-worker can run at any time and such h-workers are said to be single instance h-workers.

Most of these tasks are never done, or only done for a particular moment, this means that they must run at regular intervals. For example, the member integrity checker runs with 5 parallel instances every night between 2 and 6, while the photo email import has only one instance running every 5 minutes throughout the day.

2.1.2.1 Parallelization data access

In order to coordinate parallel data-access, the h-worker uses a simple algorithm. This algorithm builds on the fact that the work can be divided into chunks which can be identified, accessed and retrieved using a global identifier.

This algorithm uses locking, to retrieve and raise a global identifier that indicates the next chunk of work. After the identifier is retrieved the h-worker instance starts retrieving the work that needs to be done.

This simplistic method has some potential problems: if the data can not be retrieved at that moment, it is skipped. Depending on the task at hand this could be a problem and in such cases more complex methods should be employed.

2.1.3 Implementation

The hyves-daemon system is responsible for setting up the h-worker according to specification. If a task indicates that it needs to be installed on 3 servers with 5 instances per server then the hyves-daemon system will install 5 daemons for this h-worker on each of the three servers specified.

Each daemon is a small bash[1] script started by default when the system boots. This bash scripts execute the h-workers and monitors the instances to see if they exit. After an instance has quit, the bash script will sleep for a while and then re-execute the h-worker. The time the bash script sleeps is dependent on on whether h-worker quit without doing any work or not.

[1] The standard shell environment used in linux

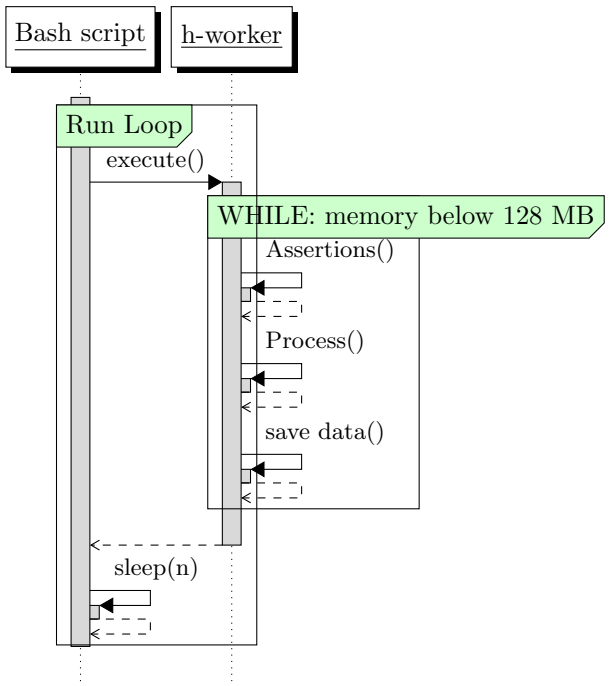

Figure 2.1: H-worker Execution

This implementation allows a task to be executed perpetually while the execution of a h-worker stops after it has performed its sub-task. When a task is only supposed to be executed between certain hours, a small assertion at the beginning of the execution makes the h-worker quit before it performs any work outside the designated hours or when the work has already finished in this period.

After the execution of a h-worker passes the assertions it will start by retrieving a chunk of work for processing. The reason that the h-worker quits after doing the work just to be restarted by the bash script, is to work-around a memory-leak in one of the used software libraries. This work-around has been optimised so that the php script does not exit until it reaches a certain memory footprint. This reduces the overhead of frequently restarting the php script.

See figure 2.1 for a graphical representation of the h-worker execution sequence.

2.1.4 Resource usage

The number of parallel h-worker instances needed is roughly determined by using the following formula:

$$endtime = starttime + amount\_work/(througput * nr\_hworkers) \quad (2.1)$$

endtime indicates the moment at which the task is finished, *starttime* the moment at which the task is started, *amount_work* indicates how many units of work there are to be processed, *throughput* specifies how many units of work

an h-worker can process per time unit and the $nr\_hworkers$ is the number of of h-worker instances running simultaneously.

Because some of the variables are unknown, this formula can only tell us that the more simultaneous h-worker instances, the sooner the work will be done, with an upper limit that is defined by the maximum number of chunks the work can be divided into. The unknown values in this formula are throughput and the desired end time which is defined as 'as soon as possible without overloading the system'.

Because the h-workers are always present in the system, an indistinct picture of the system resources that are actually needed at the moment exists. Because it is unclear how many resources are used by waiting h-workers and how many are used by h-workers that are actually running. So for scaling a task, the new number of simultaneous instances is guessed and just to be sure, it is overestimated.

No real numbers are available on the resource requirements for h-workers. It is however expected that the required processing capacity changes to the configured maximum when the tasks starts and stays at that maximum until the tasks finishes, after which its capacity needs drops back to zero. This is represented in figure 2.2. There are a few offloaded tasks that require running at small intervals or maybe even perpetually. These tasks have a continuous resource demand during the day. The needed processing capacity for a prefetching or data preparation task can thus be very sudden and demanding or continuous during the day.

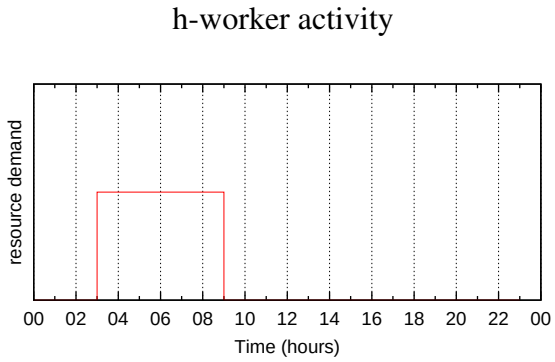

Figure 2.2: H-worker activity during a day

2.1.5 Problems

The current implementation of the h-workers and the hyves-daemon system has several problems, usually caused by the static configuration and a lack of proper statistics.

2.1.5.1 Lack of statistics

There are no numbers on throughput, resource usage per task or other measures that give insight into the current state of the system. Therefore no clear view on the available or needed capacity at any time, making decision about scaling guess work.

2.1.5.2 Static configuration

The static configuration only allows the system to be configured for peak load. This means that the number of h-worker instances that are needed during the period in which the h-worker is allowed to run will also exist during off hours. This leads to an unclear picture of the amount of resources that is actually needed at any moment, in other words: It is unclear how many resources are spent on over-capacity during peak and off-hours.

2.1.5.3 Failure resistance

When a server with h-worker instances fails, those instances are not recovered by the hyves-daemon system. This means lost capacity for the tasks that did not have enough over-capacity configured to deal with this. For a single instance h-worker running on that failed server operator intervention is required before it is restored.

2.2 Implementation of offloaded tasks

Important sections of the Hyves website depend on the operation of offloaded tasks to obtain a faster response time for the user. Performing the tasks in the foreground would in most cases significantly increase the response time of the website.

2.2.1 Tasks

Typical tasks that are offloaded to the back-end are:

- Sending email notifications for new messages, photo comments, etc.
- Sending sms notifications.
- Processing page hits for statistics.
- Member deletion, cleanup of more complex data.

For a complete list of offloaded tasks, see section A.3 on page 58

2.2.2 Technique

Offloaded tasks are implemented using the unbounded buffer producer/consumer design, discussed in [31].

This is achieved by lifting the producer/consumer design to the level of distributed programming. In the producer/consumer design, two processes are described that share a common buffer. The (first) a producer produces pieces of data and stores these in the shared buffer, the consumer then consumes the

data from the buffer and processes it. At Hyves the shared buffer is actually an external database which acts as an unbounded buffer. Just to keep the terminology consistent with that used at Hyves, the terms queue and buffer are considered interchangeable. Because communication with the database happens using the network, the producer, consumer and buffer can be on different servers.

The front-end acting as a producer compiles all operands required to perform the task and insert it into the proper queue, with each offload task having its own queue. The set of operands needed to perform a task is called an event, so each event represents one execution of the task. The back-end forms the consumer and executes the function with the parameters retrieved from the queue.

Each task has its own program to function as the consumer for that task, this consumer will, on execution, fetch a number of events from the queue and process them. As with an h-worker the execution of a consumer is said to create an instance of that program and these instances can run in parallel to increase to processing capacity for a task.

The queue itself also uses system resources and to increase the number of events a task can handle it is necessary to create multiple queues for the same task, we call this partioning. Each of these queues is a segment of the whole queue, this setup is called a 'distributed task'. A task that has only one queue is called a 'non-distributed task'. See figure 2.3 on the facing page for a graphical representation. This leads to the producers and consumers needing a way to be distribute the putting and pulling from the queue equally over all segments. The producers use a round-robin method to balance their inserts over all segments, and the consumers are statically divided over the segments. Other algorithms akin to those in load balancing are possible, see [39].

2.2.3 Implementation

The queue structure is implemented on top of a MySQL database (see section A.4 on page 59). This is done by defining a database table in which the events can be saved. Using the SQL language over the network, events can be inserted and retrieved from the databases. The communication take place over tcp/ip or unix sockets, depending on the location of both end nodes. Because each task has its own table in the database, multiple queues can coexist in the same database instance. See section A.4 on page 59 for more information about databases.

This implementation was chosen because MySQL was, to Hyves, a proven technology and therefore cut down on development time considerably. Apart from this, the database also provides persistent storage in case of failure and the ability to implement priority, weighted and other types of queues by redefining the calls and table definitions. However it is known among the Hyves team that a database is unlikely to be the most efficient implementation for queues.

All tasks with non-distributed queues have their queue placed on the same node and their consumers run from a set of servers that connect to this 'queuemaster' node. Each task with a distributed queue has a set of dedicated servers, each server holds a queue segment and the consumers that are statically as-

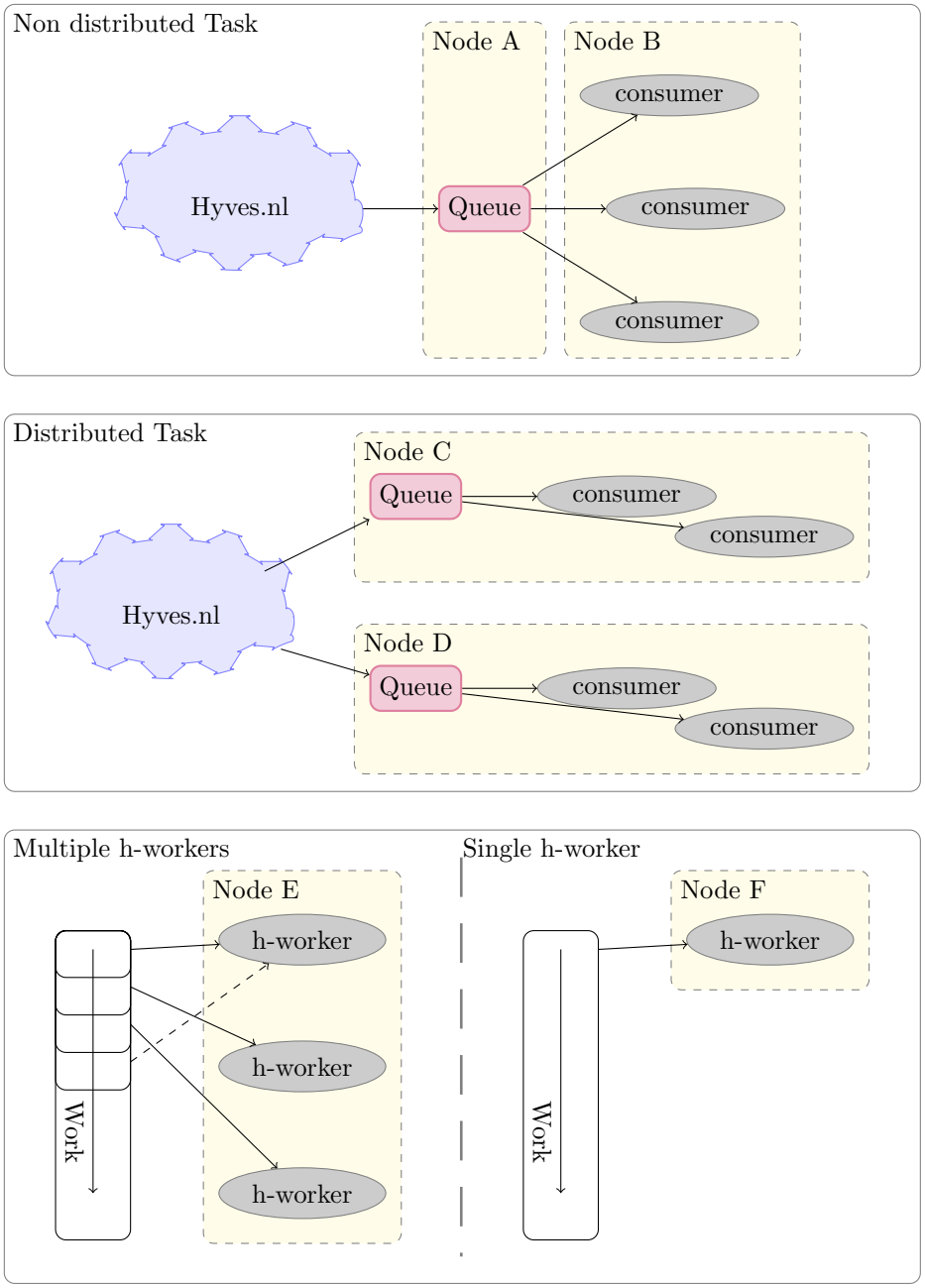

Figure 2.3: Abstract design for consumers and h-workers currently in use by Hyves

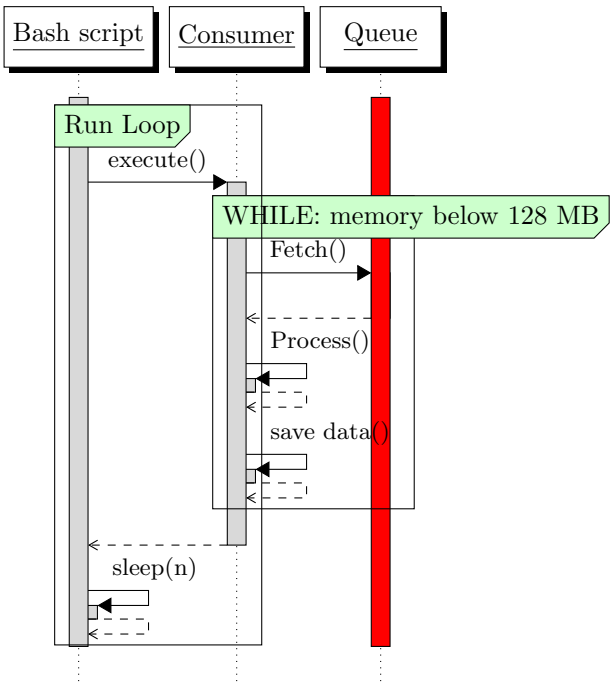

Figure 2.4: Consumer Execution

signed to that queue segment.

The execution path for a consumer is basically the same as for the h-workers, except that it fetches a number of events from its queue (segment) and starts processing those. See figure 2.4 for a graphical representation of the consumer execution.

2.2.4 Resource usage

To determine the number of consumers required, when processing the events synchronously, we can use the following formula:

$$peak\_insert\_rate = throughput * nr\_consumers \tag{2.2}$$

peak_insert_rate is the event insert rate during peak hours on a queue (segment). *throughput* represents the number of events processed by a consumer, based on the same time period as the *peak_insert_rate*. The *nr_consumers* is the number of consumer instances running simultaneously. This formula assumes a synchronous system where the events can not be kept waiting.

This applies to all queues, distributed or not, but it also applies to each queue segment in a distributed queue. Even though there might be enough consumers in the system, when one queue segment has too few consumers that queue segment will create a backlog of events that need to be processed. In the current situation however, this formula is unusable because both the throughput per consumer and the peak insert rate are unknown in the live system. The current approach therefore is to start an excess amount of consumers to process the queue at peak periods and leave them running all day and start

more when a backlog of events is detected. This system depends on the built-in sleeps to limit resource usage during off hours.

No real numbers are available on how the insert rate behaves during the day. It is however expected to follow the same curve as seen when measuring website usage, see figure 2.5. This means that the needed processing capacity per queue will also follow this pattern, and will change gradually over the period of a day.

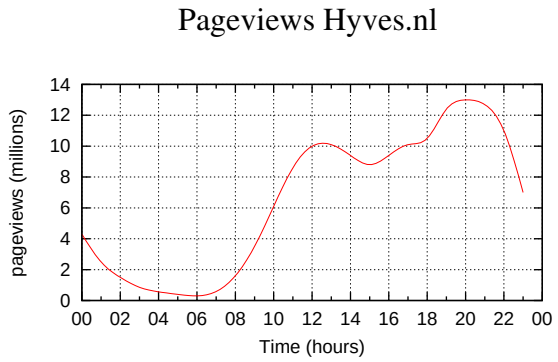

Figure 2.5: Typical number of pageviews during a day

2.2.5 Problems

Much of the problems for the offloaded task setup has great overlap with the ones experienced for the pre-fetching and data preparation tasks. There are some distinctions though as will be discussed below.

2.2.5.1 Lack of statistics

The numbers on event insert rate per task are hard to get, there is currently no way to retrieve per task insert rates when queues are located on the same server. This means its difficult to tell the difference between overcapacity or -just enough- capacity for most of the offloaded tasks. Under capacity is luckely easier spotted because the queues start filling up.

The lack of statistics make it easy to over-commit the resources of a server that is used by multiple tasks, be it to run h-workers/consumers or queues. Determining which tasks are using the majority of resources on an overloaded server is difficult and challenging, better statistics would improve this situation.

2.2.5.2 Failure resistance

For tasks that use a non-distributed queue, these queues are grouped on a single machine ('queuemaster') for maintainability. This means they are sensitive to failure of that single machine. Using manual fail-over the system can be switched to use a cold-spare that is available. The tasks with a distributed queue are better protected in case of failure. Failure of one server will mean

an increase in load for all other nodes, but will not lead to a failure of the task as a whole. For both systems, events might be lost in case of (partial) failure but this is defined as acceptable.

2.2.5.3 Static configuration

The static configuration poses the same problems as with the pre-fetch and data preparation tasks.

2.3 Existing techniques

The last two section explores two currently available techniques, namely virtualization and batch systems. The purpose of this exploration is to decide on their usefulness in a solution.

2.3.1 Virtual machines

Native virtualization is one of a few virtualization techniques that are broadly applied. Others include operating system virtualization and application virtualization. The main difference between these three types is the level on which virtualization is applied.

Application virtualization encapsulates an application, thereby abstracting it from the hardware and operating system. Examples of this principle are found in the Sun's Java Virtual Machine [17] and Microsoft's .NET framework [22]. Operating system virtualization, applied in, for example jails, [15], is often used by hosting providers to give customers seperate production environments, while avoiding the overhead of running a (possibly virtualized) server for every customer.

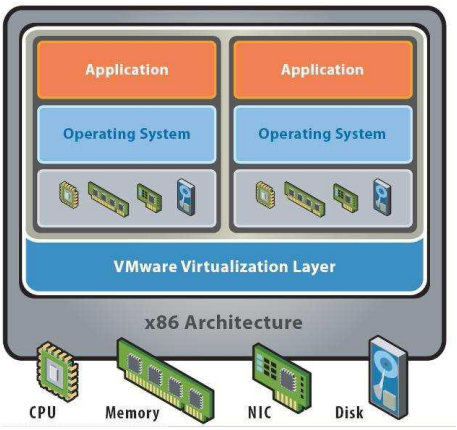

Figure 2.6: Virtual machines simulate a hardware environment [36]

Native virtualization provides a complete virtualization of the physcial hardware, and basically packages the operating system, filesystems and installed programs in a container called a virtual machine (VM) (see figure 2.6).

It is this packaging of an entire operating system that might be of interest in a possible solution.

Native virtualization solutions are offered by a number of products, most notably Vmware[36], Parallels[28] and Sun virtualization[33]. A small inventory was drawn up during the project, to establish the capabilities of the different products and have a look at other aspects such as licensing, maturity and future developments. VMware offers at least the same set of functionality as most other mature products. Also, there is the convincing fact that there is already in house experience with the product. When a virtualization product is needed in the proof of concent, VMware will be used. Further elaboration on virtualization is also based on VMware. A more complete survey of virtualization producs should be made at a later stage, when it is clear that virtualization will be used in the final application.

By simulating a complete hardware environment, the native virtualization solution can provide each VM with the same virtual hardware platform (see figure 2.6 on the facing page), while the real hardware might contain a variety of platforms. This abstraction allows for several advantages. First is the possibility to run multiple VM's on one physical node. This is a popular use of VM's as it allows the consolidation of several physical systems onto one system, which of course must have the capacity to acommodate this.

Another advantage is that a VM can run, unaltered, on top of virtualized hardware, making hardware diversity less of an issue.

Still, in this case the most significant advantage is on-line migration of VMs between nodes, allowing the movement of a virtualized system from one physical node to another without downtime. This allows load balancing VMs across a set of real servers, making sure that each VM gets the resources it requires.

A crude, but effective, high availability method is also implemented on several of the products mentioned above. This method works by restarting the virtual machine from a shared storage when it is detected to be down.

To support live migration, load balancing and high-availibility, all products mentioned above require a shared storage facility that can be accessed by all real servers to store the virtual machine. Everything comes at a cost, and so does virtualization. This is demonstrated in lost performance when compared to running on bare hardware. [19] measures overhead to be less than 6% for CPU intensive workloads and up to 9.7% for I/O intensive workloads. A more complete study of the overheads caused by virtualization and the reasons for them is presented in [2].

By design, a VM can not exceed the hardware limits of the system it runs on. The smaller VM's are in relation to the hardware, the more effective load balancing can take place. Also, the consolidation factor will then be much larger. When a VM becomes too large, these benefits are lost while the disadvantages remain. A VM that needs an entire node to itself will in any case, but not exclusively, fit the definition for 'too large'.

2.3.2 Batch system

Batch systems, also known as distributed job schedulers, are often used in the scientific or industrial world to provide computational power beyond the limits of a single machine. The system usually manages the resources provided by

a set of hardware nodes, called a cluster. The system also manages a list of tasks that have some resources provided by the cluster. The available resources are then mapped to the requested resources in order to run the list of tasks as efficiently as possible.

Typical tasks run in a batch are cpu or io intensive applications that need to search a large space of possibilities. These applications can often be run in parallel, and as such benefit from the set of nodes provided in the cluster, in order to improve the time to completion.

A number of solutions are on the current market: Condor[35] developed by the university of Wisconsin-Madison, SGE[10] by Sun, moab[24] by Clusterresources and maui[20] provided as an open source alternative to moab. An inventory of these products was drawn up and can be found in section A.1 on page 55. Condor provides the most complete set of features and the most intuitive structure. While it might not deliver the best performance this was deemed less important than mature fault tolerance and high availability methods. Condor will be used in this paper to present solutions and implement a possible proof of concept.

By managing the resources of multiple hardware nodes and scheduling the waiting tasks onto those resources, the system creates an abstraction between each task and the resources it needs. This abstraction is the key to the success of this system and allows the scheduler to create the following advantages over a set of nodes managed by a manual operator or static assignments of tasks to resources:

Better utilization of the available resources is achieved because the scheduling algorithm can reassign resources to a job as soon as they become available. The scheduling algorithms differ from product to product and can vary from a simple FIFO algorithm to more complicated matching algorithms like condors[30]. A better utilization ultimately leads to faster execution of the submitted tasks.

Better load distribution is achieved because the system knows the state of all its resources and manages them in an attempt to achieve optimal use. This includes using all resources whenever possible, and depending on the system and algorithm it might also include migrating tasks in order to optimize resource consumption.

Tasks that fail can be configured to be restarted automatically, thus a rough failure recovery for tasks is achieved.

Scheduling can take into account tasks or node requirements and preferences, allowing tasks to only execute on nodes with certain software or to have a preference for certain nodes. This becomes useful when the nodes in the cluster are not uniform in terms of hard- or software. Jobs can also be given priorities, so that higher priority jobs will be assigned resources first.

Because the set of tasks is managed by the batch system and jobs will only run when there are resources, it is easy to submit a weeks worth of tasks. This creates a backlog of tasks for the system which allows it to utilize the hardware to its fullest and makes it possible to submit tasks now without overloading the system.

This has several disadvantages, one of them being that strict control over which task is run at which moment is relinquished. Immediate execution of jobs is no longer possible, because the scheduler must first match the job to available resources. The moment at which a job will run has become a function

of the algorithm and parameters like available resources and priority. In a busy system tasks that are submitted now, might run in a few days. Another disadvantage is that the operational aspects of a system like condor have a steep learning curve.

Chapter 3

Proposed Solutions

During the research and inventarisation of the current situation, two possible solutions were developed that, at least on a conceptual level, answered to all problems and focus points. Based on the requirements (section 1.3 on page 6 and section 3.2 on page 23), one solution was chosen and further developed.

The first two sections will cover the made design decisions and set design goals. The third section will briefly discussed both solution while the fourth section reaches a decision on which solution is further developed into a proof of concept. The favored solution is discussed in the fifth and final section while the rejected solution can be found in the appendix (section A.6 on page 60).

3.1 Design decisions

A number of design decisions were taken in advance to limit the potential complexity of the resulting system in terms of maintainability and development time. These decisions boil down to the way in which choices are made and whether the system is event or time based. Both decisions are motivated below.

3.1.1 Global decisions

From the start, it was clear that the solution had to be a distributed system, as is the original (system). Every distributed system has to make a mixture of global and local decisions. A good example of such a global decision is whether the system still has enough capacity.

3.1.1.1 Centralized decisions

A common way to make centralized decisions is to assign a master which takes all decisions and informs the nodes in the system. Global decisions are easily made in such an algorithm but require that a master is selected or assigned. In case of failure the possibility of recovering system state is needed. Another potential problem is that such a centralized approach might require more resources than a single node can deliver.

Master selection One of the requirements of a central decision algorithm is that a master is selected. To pass the failure resistant criteria dynamic selection of the master is necessary, a new master should be select in case the current

master fails. A good example is the election of a root switch in the spanning tree protocol [14].

However dynamic selection algorithms can select multiple masters in case of a network failure that splits the distributed system[1] in two or more sections, a so-called split-brain situation. This poses difficulties for the single-instance h-workers.

Master state Re-selecting the master in case of failure raises the question of preserving the master state and whether this is necessary. Depending on the ability of masters to collect the state of the system and the importance of this state, it might be necesary to preserve master state during fail-over.

Common ways of achieving state recovery in case of failure are replication, replaying log files or checkpointing (see [34]).

Scalibility This centralized approach creates a possible bottleneck on the master because it must keep and update a system wide state, perform all centralized decisions and inform the necessary nodes. Depending on the complexity of these tasks and the size of the system, required resources might exceed the limits of a single node.

3.1.1.2 De-centralized decisions

Global decisions can also be taken in a de-centralized manner with multiple nodes coming to the same conclusion. To achieve such a system, two options are available: Either all systems must have the same information at the time of the decision, thereby making sure that the outcome is the same on all systems, or consensus must be reached afterwards.

Timing In both situations timing is essential, to ensure that the decision process is started at the same time on all participating nodes. Many clock synchronization algorithms are available [[5], [7], [16]]. However ntp ([23]) is in widespread use and should be used whenever possible.

However problems which are the result of bad timing are notorious for their elusive nature, resulting in potential problems that are difficult to solve.

Consistency Ensuring that all systems are consistent (have the same information) when they make a decision ensures that the outcome of that decision is the same on all systems. A lot of consistency models exist, from very strict to lazy, with or without synchronization operations. From the models discussed in [34], strict consistency is the most applicable.

Consensus Instead of guaranteeing consistency, the distributed system could also try to reach a consensus over the outcome of the decision. The assumption is that without consistency restrictions and just a best effort guarantee to send state changes to all other nodes, the majority of the systems will still have a consistent view of the data. By comparing the decisions made and reaching a consensus on them, the correct decision will be taken.

[1] D.s. al uitgelegt

This could be accomplished with a majority vote, after all nodes have exchanged their results.

3.1.1.3 Conclusion

While for most of the problems described for both the centralized and decentralized decisions, well working/flexible solutions can be found in literature, implementing such features is complex and error prone. Self-built solutions would, in such cases, only add to the workload of the engineering teams and are therefore unwanted.

To avoid such problems, each node takes global decisions, but a random factor makes sure that only a small amount of those decision are really effected.

3.1.2 Event or time based system

The proof of concept, like any system cab be event or time based, or a mixture of both. Time based systems are programmed to take actions that are triggered by the passing of time. Event based systems take actions based on received or triggered events.

Because actions do not allways have an guaranteed outcome an event based system must have the ability to re-trigger events after a certain period of time. This raises the question of specifying an effective timeout value.

The time based system takes actions at a defined interval. If these actions do not have the desired effect they will be taken again the next interval.

Time based systems are simple and predictable by nature, and as such the proposed solutions are time based to improve the design and development time.

3.2 Design goals

To reach the goals set in research focus and solve the main problems, a number of goals have been set that should be reached in these designs.

- automated scaling of consumers and h-workers
- providing a clear view of used resources
- better monitoring options
- failure resistance
- load balancing
- providing facilities for single instance h-workers.

3.3 Solutions

The first solution, called A, creates a container type for each task. These containers contain the elements required to perform the task: a set of consumer instances and a queue segment for all offloaded tasks and just a set of workers for the other tasks. A task is thus performed by its set of containers. The number of instances per container is fixed, so that each container has a maximum throughput. The containers are executed on a set of servers, when a single

server becomes overloaded some of the local containers are moved to another server. When the number of containers for a task is not enough the containers will indicate this and more containers will be started to create enough processing capacity. These increases happen gradually ensuring good scalibility for the offloaded tasks, pre-fetching and data-preparation tasks however have more ad-hoc capacity needs.

The other solution, called B, clearly separates the queues and processors/tasks and scales each separately. The queues are split up into a minimum of two queue segments (containers) and then these containers are divided over the available nodes, distributing incoming events between the containers. Containers are further split up and divided over the nodes when cpu usage indicates that the node is too busy. For each container (queue segment), a manager is run in a batch system. This manager monitors the queue length and starts workers that process the events stored in the queue segment. For pre-fetching and data-preparation tasks, that do not use a queue, a single manager is started in the batch system. This manager knows how much work needs to be done and starts workers to do the task. Because these managers and workers all run on a batch system, the processing capacity can be increased on demand, either gradually or ad-hoc, to be determined by the manager.

3.4 Choice

Although the second approach this seems more complex, it seperates the queues from the processing, a seperation of concerns. This results in a system that clearly allows for distinct scaling of the queue and consumer/h-workers to their own needs. This creates optimal conditions to answer to the h-workers' ad hoc resource demands. All in all, this makes it the most flexible of the two systems.

Separation of the concerns allows a distinct insight in used resources for each of the concerns, thereby meeting one of the main design goals. This separation also allows fault isolation between the concerns, further improving maintainability.

Single instance h-workers are also an better fit in solution B and an additional benefit comes from the fact prioritization and software/hardware demands of jobs are possible in a batch system.

Solution B is chosen to be further developed and tested.

3.5 Solution B

As the chosen solution, solution B will be explained hereafter. This is done by discussing each concern in turn, and then exploring them by listing the key concepts and exploring how the design goals (see section 3.2 on the previous page and section 1.3 on page 6) are met using these concepts.

3.5.1 Queue concern

The first concern is load balancing and scaling the queues on the system. The goal is to use a minimum amount of hardware while keeping the load on each node acceptable. By influencing the distribution of incoming events among the available nodes, the systems slows down insert rates on busy nodes in an

attempt to lower the load. Besides nodes can be taken in and out of use according to the total capacity needs.

3.5.1.1 Node

A node is hardware that is available for use by the container described in this section. Each node has a state indicating its availability. This state is determined by the resource consumption of the database and the containers present on the node. Available states are as follows:

IDLE has no containers, is not used by the system

NORMAL has a normal workload

BUSY node is busy and load balancing can not move work to this node

TOOBUSY node is overloaded and load balancing tries to move work to normal or idle nodes

At a certain interval, each node takes a set of local decisions. These are based on local information and the information concerning all other nodes available through the global state. These decisions are further described in the sections 3.5.1.4 to 3.5.1.7.

3.5.1.2 Container

As mentioned, each queue is split up into queue segments, every one of which together with a weight forms a container. Each container belongs to a type, according to the queue it represents. The minimum number of containers that must be available in the system can be set per container type, allowing to prepare the system for high throughput queues on start up and after scale down (see section 3.5.1.5 on page 27).

Each queue has 1000 weight points that are divided over the containers. The distribution of these points determines how incoming events are divided over the containers. In order to balance queue loads over multiple nodes, the weight points can be partially (or fully) moved to other nodes.

Each node keeps a list of containers it holds. On receiving weight for a container it checks whether it already holds the container, if so it will raise the weight of this container. If not, a container of the required type is started.

Implementing start and stop methods for a container allows the container to perform a number of actions on start up or shutdown. These actions can include checking/cleaning up the environment or starting/stopping a manager in the batch system.

3.5.1.3 Global state

The global state is a fictional layer holding state information about all nodes. This information is tapped in order to facilitate localized decisions regarding global matters, like scaling hardware as discussed in 'scaling up/down' section 3.5.1.5 on page 27.

At regular intervals each node sends an update to all other nodes, which then update their global states accordingly.

To guarantee freshness of this information a timeout is defined. When a node stops sending updates it is eventually considered offline and removed from the global state.

The update consists of the following information: node name, node state and the list of containers it holds.

Communications for the global state take place over multicast; this allows the node to send a one to all message. This is far more efficient for the sender than sending N-1 messages using normal tcp or udp mechanisms, where N is the number of nodes in the system.

Using multicast also allows a flexible membership system. No list of participating nodes is needed, a message sent using multicast is received by all systems listening for it. When the system spans multiple network segments, multicast routing must be employed on the routers. Several protocols are available for this, see: [38], [25] and [1].

Sending these messages and maintaining the global state will not become a bottleneck, because the numbers listed below fall well within the capabilities of a node. The number of updates received at each interval by every node is equal to the number of nodes in the system. A modern node can process tens of thousands of packets per second[2]. The maximum size of each update is chosen as the maximum packet size, which is roughly 1500 bytes. Assuming 8 byte names for nodes and containers and 1 byte for a state. An update can contain a maximum of 186 containers[3] per node.

The memory size used for maintaining a global state on each node is small and is a function of the number of nodes and containers in the system. Assuming that the overhead of storing the containers and nodes in memory doubles their memory requirements, due to overhead introduced by the used structures, we can store roughly 7000 nodes or containers in 1 MB of memory.

3.5.1.4 Load balancing

One of the local decisions taken by a node is whether load balancing is needed. When the node is TOOBUSY, it selects a container. The weight of this container is split, one half remaining on this node and the other half is moved to a selected target node.

Selection of the target node follows two steps. The first step tries to find a NORMAL node already holding the container. If this fails the second step is to select just any NORMAL node. Preference is given to nodes already holding the container to avoid splintering a queue into many containers. If no NORMAL node is found the target node is selected from the IDLE nodes. This is further explained in section 3.5.1.5 on the next page.

In order to distribute the incoming events over the available containers, a load balancer monitors the weight of each container on every node. This load balancer directs incoming events according to the weight of each container. The used algorithm is called Weighted round robin and is further explained in section A.5 on page 60.

The load balancer is a potential bottleneck in the system. The setup works by using a pull mechanism. The load balancer must monitor every node for every possible container regularly, which amounts to N x C checks each interval.

[2] a quick estimate: 1Gb/s divided by 1500 bytes is 83.333 packets/s

[3] $(1500 - nodename - nodestate)/8$

N represents the number of nodes and C the number of possible containers. The number of check could raise rapidly when the system grows, potentially overburdening the load balancer with checks.

The interval at which these checks are done must be chosen wisely and is a trade off between reaction time and the resources required do these checks.

Moving weight between nodes is relatively light work, a simple request and reply indicating success or failure suffice. The request only needs to hold the container name and the weight that is moved. On success the source node lowers the weight of its own container and the target node raises the weight of its container or starts a new one. On failure everything stays the same. These messages require one to one communications and can use tcp or udp.

3.5.1.5 Scaling up/down

The system attempts to use as few nodes as possible. To indicate that they are unused nodes change their states to IDLE. Nodes that are in use, have the NORMAL, BUSY or TOOBUSY state.

As part of the decision cycle on each node, it is determined how many of the used systems are normal and how many are BUSY or TOOBUSY. These numbers are used to determine whether there is enough processing capacity in the queue system. When the majority of systems is BUSY or TOOBUSY it is determined that unused nodes can be used as target nodes when balancing the load. This effectively enlarges the number of used nodes when the system becomes busy as indicated by the BUSY and TOOBUSY state.

When the majority of used nodes returns to NORMAL, there is over capacity in the system and it will try to scale down the number of used nodes. When this condition is detected in the decision cycle, a chance will determine whether the node will try to offload all of its containers to NORMAL nodes.

Offloading is done by moving all the weight from all containers to another node, effectively returning to the IDLE state. During the offload process, it is ensured that the minimum number of required containers stay in the system. When a container can not be moved because the minimum is reached, the offloading process is aborted.

This aspect of the system allows the system to adjust the number of used nodes to the current requirements. This also makes sure that containers adapt to changing load conditions from day to day. In future versions of the system the possibility of using the spare capacity of IDLE nodes for other purposes can be explored. The nodes might also be shut down to save on power consumption.

3.5.1.6 Failure resistance

There are several failures that can occur in such a system, the most common of which is the loss of a node. This could be because of communication failure, bugs in the software, hardware failure, etc. Regardless of the cause it is assumed that this is experienced by loss of the ability to communicate with other nodes.

In such a case, the global state updates will no longer be received by other nodes and the node will be removed from the global state as soon as the freshness of the last update expires, effectively removing the node from the system. The node is reintroduced into the system as soon as its updates are received again.

The load balancer notices a failed node because the checks, discussed in the previous section, fail. It responds by directing incoming events to the remaining containers. Because each queue has a minimum of two containers there is always at least one container left if a node fails.

Depending on the failure the node might lose its state. The containers and their weight on that node are then lost to the system. Because each weight point represents a share in the total, only the granularity in which load balancing can take place is influenced. This is no problem as long as a fair portion of the original points remain in the system. The loss of a container however, might drop the number of containers for that type below the minimum, which potentially cancels the fail-over capabilities for that container type.

When the node does not lose state, for example because the problem was network related, the node rejoins the system as soon as communication is re-established, with its containers and weight the same as when it left the system. The load balancer detects this, because the periodic checks succeed again, and restarts sending events.

The final version of the system is to detect the loss of a node, and insert a new container when the number drops below the minimum. The system should also attempt to keep the total weight per container type at 1000, anticipating failing and rejoining nodes.

3.5.1.7 Monitoring

Because the queue system is to be developed in house, it can be made to work smoothly with the monitoring systems in place. An interface could be provided to pull status information from nodes or the system could push information to monitoring systems. Available information includes system resources used by the database, the list of present containers and their weights and the managers associated with each container.

3.5.2 Consumer/h-worker concern

The second concern is scaling for the consumers and h-workers according to their needs. The goal is to make sure that events are processed in time for each consumer and that all offloaded tasks are executed in a timely manner. Key to achieving this goal is the implementation of consumers and h-workers in a manager/worker system running on top of a batch system. By giving a central authority - the manager - the responsibility of monitoring the work at hand and spawning workers to do the work, it is easy to scale the processing capacity up and down - by starting more or less workers - according to capacity needs. A manager will quit only when the task is done.

This all runs on a batch system, taking care of distributing the workers and manager over all available resources.

3.5.2.1 Manager

There are two kinds of managers in the system: the managers that are associated with a container (queue segment), and the managers that perform offline tasks. While each of these has the same responsibility, making sure that the work gets done, the implementations differ.

Each container (discussed above) starts a manager in the batch system. This manager monitors the associated queue segment. The term queue manager is used to refer to this kind of manager. The manager keeps a worker insert rate that defines how many workers are started every x seconds. By monitoring the queue length of a segment, it is possible to determine whether the processing capacity is sufficient and the worker insert rate is raised or lowered accordingly. Raising the insert rate is done when the queue length exceeds a threshold, indicating that events are not processed quickly enough. Lowering the rate is done when no waiting events have been detected for a short period of time. This results in an algorithm that keeps adjusting the insert rate towards the optimal value for that moment. Because the insert rate is assumed to vary over time this is acceptable.

For the h-worker managers (task managers), there are several possibilities depending on the data that is available in the system. When for example it is known how much work needs to be done and how much one worker can process, all needed workers can be submitted into the batch system at once. The manager only needs to wait until all are finished and check whether the work has truly been done. Other possible algorithms include guessing and a steady insert rate of workers until the work is done.

All these algorithms could be complimented by using a feedback loop of historic data from previous runs to guess the needed capacity more accurately. The queue manager could use trend information about insert rates from prior days, while the task manager can use the history to more accurately guess the number of needed workers. This is however designated as future work.

3.5.2.2 Worker

The workers started by the manager are a small variation on the old consumers and h-workers discussed in 'State of Art' chapter 2 on page 7. The bash loop is removed in both cases, so that the new workers are certain to exit at some point. This however means that queue managers must constantly submit new workers, in order to obtain a steady availability of processing capacity.

The alternative, allowing workers to run perpetually and implementing a mechanism for the manager to stop running workers, is far more intrusive and prone to failure. When a manager crashes, it loses state. Running workers can therefore no longer be stopped.

3.5.2.3 Batch system

The batch system consists of a set of execution nodes which together represent all available resources. These nodes are all configured statically and are not released during off-hours. The batch system has to provide a number of facilities for successful operation. First and foremost the batch system has to distribute the submitted managers and workers (called jobs) over all available resources. Additionally, it has to restart jobs when they have finished abnormally, for example due to a crash or node failure. Finally, a crontab like facility to execute task managers at designated times, if these managers were not already running, must be available.

3.5.2.4 Load balancing

It is the batch system's responsibility to distribute the jobs over all available resources, making sure that these resources are used as effectively as possible.

3.5.2.5 Scaling up/down

The managers are responsible for scaling processing capacity up and down, by submitting the proper amount workers. The batch system is in turn responsible for executing the submitted jobs, while taking into account available resources and priorities.

The batch system itself does not scale up or down, but the utilization of the available resources does drop during off-hours. Future work might include using free resources by other lower priority tasks to make optimal use of the available hardware at all times.

3.5.2.6 Failure resistance

The batch system can restart a job in case of failure. As a result that the manager can be considered reliable in case of failure. When workers crash it is expected that the work they were doing is lost; this is deemed acceptable. If not, the manager and workers must implement a mechanism that will retry to process work that failed.

The batch system provides its own methods for high availability, making sure that essential tasks are taken over by a different node in case of failure. During such a failure the state of the managers that crashed is lost. The managers are however designed to quickly recuperate from such crashes.

3.5.2.7 Monitoring

The managers and workers are developed internally and can be (re)written to provide statistics to the monitoring system in place, thereby giving good insight in the current insert rate and the number of consumers executing per second. Extra intelligence must be built to detect non-running managers or single instance workers that should be in the system.

The batch system should provide adequate methods for retrieving information like utilization and the number of waiting jobs. As an added benefit it would be useful to see statistics on how many resources each manager gets assigned.

3.5.2.8 Single instance daemons

The batch system makes sure that jobs submitted into the system only run once. The managers described above are all single instance daemons. The single instance daemons described in section 2.1 on page 7 should be considered managers that do not submit workers, but perform the work themselves. In case of a split brain there is the possibility that the batch system is no longer in touch with the node running the job. In such a case it might re-spawns the job while the allready running job is not stopped. There are several solutions to this: a node in the batch system should stop all running single instance ones it can no longer reachs the central manager of the batch system. Or the single

instance should check connectivity with the central manager itself, or try to obtain a global lock.

3.5.3 Conclusion

The introduction of the batch system and manager/worker paradigm allows for a system that can scale processing capacity fluently or abruptly at the discretion of the manager. This allows the managers to follow the required capacity, which is more efficient than the method currently in use, which is always configured for peak-load. The queue system is also scaled according to its own resource consumption, and using weight to distribute incoming events between available queue segments introduces great flexibility and fail-over abilities.

The queue and batch systems are clearly separated in this design, each running on its own set of nodes. In future versions these systems could be placed on the same node to allow worker or manager execution on the same node as the associated queue segment, thereby reducing network overhead. When capacity does not allow all workers to execute on the same node as their containers, they can be executed elsewhere. Both the queue and batch system need to be configured to allow for this without overloading the node.

One of the implicit goals of this research is to remove the static configuration that made the systems described in chapter 2 on page 7 so very rigid. The static configuration left in this design is limited to a static assignment of the nodes available for each system and a definition of the container types present in the system. This for example allows the system to start the appropriate manager. The load balancer is also statically configured to monitor all nodes for all containers. This configuration needs to be adjusted when either of those change. All these settings do not hamper the ability of the system to adjust to changing work loads.

For more information about the the available batch systems and the selection of the batch system to be used in the proof of concept, see section A.1 on page 55.

Chapter 4

Proof of concept

The proof of concept (PoC) is a rudimentary implementation of the chosen design. Its goal is to test the correctness and usability of the design. Before implementing the system in a PoC, a crude model was built to recognize and eliminate major deficiencies early in the process.

This chapter is divided into two sections, the first explains the queue system and the second explains the consumer/h-worker system. Each section will describe the core algorithm of the system and highlight important design issues, following with a discussion of goals, implementation and results from both the model and the PoC.

4.1 Queue system

This section explains the PoC implementation of the queue system. First, the different aspects of the algorithm are explained. Then the model and PoC implementation are explained according to their purpose, environment and results.

For an explanation about the purpose of this queue system, please reread: 'Queue concern' in the 'Proposed Solution' section 3.5.1 on page 24.

4.1.1 Algorithm

Every node runs the same algorithm, the most important aspects of which are explained below. The aspects can be categorized into three sections:

Detection Determining node state (4.1.1.1)

Decision Select containers to move (4.1.1.2) and Determine downscale (4.1.1.3)

Execution Find targets for containers (4.1.1.4), Move containers to target (4.1.1.5) and Communication of updates (4.1.1.6)

The purpose of each aspect is explained together with its model or PoC implementation and each aspect finishes with a section of suggested improvements. Wherever the model section is left out this means that there was no model implementation of this aspect. If the PoC implementation is left out this means that it was the same as the model implementation.

4.1.1.1 Determining node state

Purpose Monitor resource usage of the database in order to indicate the state the node is in.

Model solution The model did not mimic resource usage and the system state was therefore determined by the number of inserted events per second.

PoC solution Because the resource usage turned out to be highly erratic when measured every few seconds, the average over the four latest measurements is used to determine the node state.

Improvements The system works well, but when it balances around a threshold, system state changes might follow each other frequently. To remedy this, a buffer area could be implemented. For example: the state changes to TOOBUSY when resources usage passes 50% but will only change back to BUSY when it becomes lower than 35%. Another improvement might be to use a formula to smoothen the effect of the erratic resource measurements. One such formula is listed below and is currently in use for the website statistics.

$$V = \alpha * NV + (1 - \alpha) * OV \tag{4.1}$$

α indicates the effect the latest measured value (NV) will have. For example, with $\alpha = 0.5$ the measured value will make out 50% of the result, the other 50% is determined by the previous result (OV) of the function. V is the smoothend result value.

4.1.1.2 Select containers to move

Purpose When a node is TOOBUSY, the algorithm selects a subset of its containers, to be split and moved to another, not overloaded, node in the system. Effectively this will lower the insert rate on the node.

Model solution Early versions of the model selected one random container to be split and moved. Tests showed that in the case of many containers this method was too slow so newer versions move a percentage of the available containers. Please read section 4.1.2.3 on page 37 for a more in-depth explanation.

Improvements First, information like insert rates is unavailable in the live system. Therefore no estimation can be made of the load each container imposes on a node, resulting in random selection as described above. Making effective decisions about which container should be moved is therefore subjective to chance, and figure B.4 on page 69 clearly shows that this is the case. Keeping a history of the effectiveness of the chosen selection, could facilitate the development of a heuristic method for choosing containers.

Secondly, resource consumptions of the database queue implementation can be seen as the product of two parameters per container. The number of put operations and the number of get operations on each container to be precise. Lower or raise either of them and the resource consumption will rise and fall accordingly. This behavior however is not linear.

By lowering the number of puts on a container, the number of gets will follow because the manager (section 3.5.2.1 on page 28) will adjust to the lowered processing capacity needs. The employment of this cause and effect mechanism to lower the load of a node causes problems when the node gets overloaded. In case of an overload performance of the node decreases, creating a backlog of items to be processed. Therefore the manager will raise its worker insert rate, increasing strain on the system. This can be seen very clearly in section B.2.3.1 on page 127

Future manager versions would therefore do well to monitor the state of the queue system and allow it time to recover from overload by drastically decreasing processing, after which processing can continue as normal. It is better to slow down processing for a while than to overload the system and not work at all.

4.1.1.3 Determine downscale

Purpose To minimize resource usage and the splintering of containers, the system will try to detect off hours and consolidate the containers onto as few nodes as possible.

Model solution This is done by determining the states of all participating nodes. If it is determined that a large percentage of the nodes has a normal state, the algorithm will try to select one of these nodes and try to move all its containers to another node in order to reclaim the IDLE state for this node.

Improvements The outcome of this operation is random, each night a different set of nodes could stay on-line. An improvement would be to consolidate the containers onto the most capable nodes, ensuring that when resource usage rises in the morning the weaker (in terms of hardware) nodes will not be overwhelmed.

4.1.1.4 Find targets for containers.

Purpose When the system has determined that it needs to move one or more containers it also has to determine where to move them, and whether they can be moved at all. Factors taken into account are:

- minimum number of containers for queue X
- current state of all nodes
- a possible need for downscaling

Model solution Every container that needs to be moved has an associated percentage, indicating how much weight is to be moved to the target node. This is normally 50%, unless the system is downscaling and wants to move all the weight. There is one exception: when the weight points of a container after splitting breach a lower limit, then the full container will be moved in order to avoid too much fragmentation.

For every container on the list, a target node is determined using the following set of rules:

- When scaling out an Idle node should be used.
- Use a node already containing a container of this type.
- If the minimum number of partitions is reached or none of the above works, use a node that does not have the container.

In case of downscaling the attempt is aborted, if not all containers find a successful target.

4.1.1.5 Move containers to target

Purpose This aspect of the algorithm moves the container to its target node. The algorithm splits the container (according to the associated percentage) and moves the separated portion to its target host. On success, the container weight is updated on the local node, on failure everything is left the same.

Model solution The system is implemented as discussed directly above.

PoC solution The system sets up a tcp connection to the target node, communicating the container type and the weight that is moved. On success, the weight on the sending node is lowered and that on the target node is raised.

4.1.1.6 Communication of updates

Purpose To spread the current system state to all other nodes, thereby allowing them to be informed of the current state of the system.

PoC solution This is done by sending a multi-cast message regularly, holding the current state of the node. See section 3.5.1.3 on page 25 for more information.

4.1.1.7 Restoring weights of containers

Purpose As explained in section 3.5.1.6 on page 27, the system should restore weight and containers after they are lost.

Improvements This was not implemented in the model or PoC because it was not yet necessary.

4.1.2 Model

This section explains the model implementation for the queue system according to its purpose, environment and results.

4.1.2.1 Purpose

A model was created to speed up the development of the algorithm. It allowed accelerated testing of the algorithm, so that it worked correctly when implemented in the PoC.

It also allowed and simplified testing under extreme circumstances, like large numbers of nodes or containers.

4.1.2.2 Environment

The model is constructed as a program that simulates nodes and containers in order to test the algorithm described above. The runs are then displayed in plots to interpret the results and make improvements.

The model was used to test the algorithm with four possible system configurations. Each configuration represented a number of queues and nodes.

- The first configuration consisted of eight nodes and 12 queues. Six of these queues had a maximum event insert rate of ten, five of them had a maximum of 100. The last one had a maximum insert rate of 1800 events per second. This is a possible lifelike configuration.
- The second configuration is identical to the first, except it has only six nodes, which created an overload situation.
- The third configuration consisted of 50 nodes and 200 queues, each with a maximum of 100 inserts per second. It is intended to test the system on a larger scale.
- The fourth configuration consisted of 50 nodes and 151 queues. One of these queues had a maximum event insert rate of 5000, the others a maximum of 100.

These four configurations were used to test different settings of the algorithm. The tested setting influenced how many containers were split and moved when the node became TOOBUSY. The first test only moved one container at a time, but the last two tests moved 25% and 40% of the containers on a node.

4.1.2.3 Results

The first test only selected one container to move. The results (section B.1.3.1 on page 79) clearly show that selecting the container that causes the overload is rarely accomplished at first attempt. This test is only included as a reference and as an exaggerated example of how bad random selection works.

Each plot shows five graphs. In the first graph each 'line' represents the insert rate on a particular node. Abrupt drops and rises of these lines indicate the movement of a container; the more load a container represented for that node the larger the difference. The second to fifth graph give the number of nodes in the system that were in a particular state over time. These numbers are relevant because they indicate when the system is overloaded and show that the system adjusts the number of nodes it uses according to its needs.

The tests with the 25% and 40% setting worked better as they raised the chance that the right container was selected. It was however no guarantee for making optimal choices. The 40% setting was the best of the two tested, and was therefore used in the PoC implementation. This can also be seen in table 4.1 on the next page[1]. This table shows the chance that the algorithm has selected the right container after each selection. The numbers in 4.1 only

[1] $c = \sum_{i=1}^{x} \left((1-k)^{i-1} * k\right)$ shows how the chances are calculated. k is the chance that a selection is correct, while x is the number of selections that are attempted, c then returns the chance that a right selection is made after x selections. Selections are made every 15 minutes

hold when a single container is responsible for the majority of the load, the worst case scenario. The results of these two tests are shown in section B.1.1 on page 65 and are grouped by test run, first the four plots for the 25% test, then the 40% test.

As future work the algorithm could be modified to move 20% of all containers, thereby making sure that the input rate drops with 20 percent. This however was not implemented due to time constraints.

attempt	25% selection	40% selection
1	25%	40%
2	4,7%	64,0%
3	57,8%	78,4%
4	68,3%	87,0%
5	76,2%	92,2%
6	82,2%	95,3%
7	86,6%	97,2%
8	89,9%	98,3%
9	92,4%	98,9%
10	94,3%	99,3%

Table 4.1: Chance that the random selection of a set of containers selects the right container after an attempt. Attempts are done every 15 minutes when a node is TOOBUSY

4.1.3 PoC implementation

This section explains the PoC implementation for the queue system according to its purpose, environment and results.

4.1.3.1 Purpose

The PoC was developed to test the queue system in an environment that was as lifelike as possible.

4.1.3.2 Environment

The queue system consists of three nodes: joscluster7, joscluster8 and joscluster9. joscluster9 has significantly less hardware than the other two. These three nodes are dedicated to the queue system. They can submit jobs (managers/workers) into the batch system, but no jobs are executed on them.

The system will hold five container types: one large and four smaller. This is consistent with the live environment where only a few queues are considered large. See figure B.9 on page 74 for the container names and their insert rate over time.

The input stream of events is created using a query log file from the alert container. Because logging of all database requests is heavy, this practice was stopped in spring 2008, after a series of service disruptions due to the load. The remaining log files are therefore scarce and full of service disruptions. The

used log file is thought to provide an accurate pattern of the event stream except for the service interruption, which occurs after a few hours. Still the service interruption helps to test how the whole system would react in case of failure. The log file however only provides events for the alert queue and does not provide enough events to stress the PoC. To increase pressure the numbers have been multiplied. For every replayed event from the log file the PoC inserts a multitude, the numbers of which have been empirically determined in order to stress but not overload the test system. The event insert stream during the PoC test run is shown in figure B.9 on page 74.

The load balancer has been installed on another node that doubles as execution node for the batch system. The load balancer retrieves the current weight for each container by pulling a web service on each node providing this information.

The system is created using Python[29] and the Twisted[27] library. They have been chosen because of my experience with python and Twisted offers a good abstraction from networking and as such allows easy programming of network applications.

The containers are initially deployed over joscluster7 and joscluster9; the incoming event stream is enough to stress joscluster9. Because the hardware of joscluster9 is less powerful, it becomes TOOBUSY and moves weight to joscluster7 and eventually joscluster8 is taken into use. Thus, both load balancing and upscaling to meet resource demands are tested.

4.1.3.3 Results

Scalability In order to meet the scalability requirement, the system is expected to scale according to needs. In case of the queue system this means redistributing containers in such a manner that no system stays TOOBUSY.

Figure B.10 on page 75 clearly shows that the algorithm starts shifting weight when joscluster9 becomes TOOBUSY. However, the plots shown in the overload section (section B.2.3.1 on page 127) show that this shifting is not always perfectly successful. This has been discussed in various places already, see section 4.1.1.2 on page 34.

The random factor that is applied when load balancing, can be a serious problem. Still, when by chance the random selection hits the spot, the system works quite satisfactory as can be seen in the PoC test run (section B.1.2 on page 74).

The PoC test run shows that around 16:00, when all systems are BUSY or TOOBUSY, the third node is taken into use. It is taken out of use again by the scale down that takes place during the service disruption in the replayed event stream. The same happens around midnight and 7:00, but now joscluster7 is taken out of use.

This shows that the random factors involved when choosing which containers to move, or where to move them are less than ideal.

Failover Failover is achieved by always having a minimum of two containers per queue in the system. If one fails, the load balancer will detect this and direct all connections to the remaining container, as shown in section B.1.3.2 on page 84

Modification of existing code Scaling the queue system requires no code modifications, just the extra code that needs to be run on the queue nodes. However systems that depend on the queue system, for example the consumers, need to be adapted. This is discussed in section 4.2.3.3 on page 47

Automation The new system proves to be an automatic scaling tool which makes manual adjustments virtually unnecessary as long as the right containers are moved. Due to the random selection however, the system might not run as smoothly as section B.2.3.1 on page 127 suggests and manual intervention might be needed to resolve such situations.

Efficiency Scaling down the number of used nodes works well, see for example figure B.10 on page 75. However, the set of nodes that is used during the off-hours is unpredictable as explained in section 4.1.1.3 on page 35.

To improve efficiency the unused (IDLE) nodes could be used for other tasks during these off hours, or with some future work might even be shut down during the night to save on power consumption.

Prioritization at overload Due to time constraints no system has been implemented to handle priority in overload situations.

Monitoring In the PoC implementation statistics have been gathered by letting the algorithm write periodic status messages to a database, used to created the plots found in chapter B on page 65. This is exemplary for the ability of the system to export performance data to our in-place monitoring systems.

Minimizing network overhead To minimize network overhead the queue nodes could also serve as execution nodes in the batch system. In this case it is important to make sure that the batch and queue system do not overload the node. See section 4.2.3.3 on page 49 for a more thorough explanation of how workers are run on the right node.

Load balancing As said in the scalability section (section 4.1.3.3 on the preceding page), the load balancing is well displayed in figure B.10 on page 75, for a more detailed look per node see figure B.11 on page 76, figure B.12 on page 77 and figure B.12 on page 77.

4.1.4 Conclusion

As the PoC implementation showed, the system is not ready for deployment in its current form. While the redistribution of weight works well it is just a tool, and could equally improve or worsen the situation when no correct decisions are taken about the allocation of containers.

To improve the system a more effective selection method needs to be developed. Suggested methods include: heuristics based on history, improving the available statistics in the live system or avoiding the selection process by moving a small percentage of all containers on a node.

The distributed nature of the system improves features like fail-over, but also makes the system more complex. For example detecting lost weight or lost container types in a system without a central authority requires making sure that not all nodes try to re insert lost weight or containers at the same time.

The system needs more work before it becomes usable. A better approach might be to reconsider the use of databases as a queue system. Another queue implementation might not only improve efficiency, but might also improve the starting point from which a scalable solution can be built.

4.2 Consumer/h-worker system

This section shows that the developed manager/worker system as explained in section 3.5.2 on page 28 fulfills its purpose; it can process incoming events in a timely manner while making the system adaptive to changing load conditions.

First, the purpose of the manager/worker system is explained, after which the core algorithm is presented and important sections are highlighted. As was the case in the previous section about the queue system, a small model was made to check early feasibility of the algorithm. These two, model and PoC implementation, are discussed according to their goals and results.

Section 3.5.2 on page 28 discusses the application of the manager/worker paradigm to solve the problems for the consumers and the h-workers (as explained in chapter 2 on page 7). The principles that make the manager/worker system an adequate solution, apply to the consumers as well. Also, the demands for the consumer system are the most complex and demanding:

- Deadline demands for the consumers are much stricter than for the h-workers.
- The consumer system must be able to run 24/7, while the h-worker system works at intervals.
- Determining the workers that need to be started is far more simple for the h-worker system.

Given these points and the fact that timing restraints on this research the feasibility of the manager/worker system for h-workers is assumed to be proven by the feasibility of the consumers (discussed in this section). The proof of concept system thus only implements the consumers. For an explanation about the purpose of the consumer/worker system, please reread section 3.5.2 on page 28

4.2.1 Algorithm

Every manager runs the same algorithm, the most important aspects of which are explained below. The purpose of each aspect is explained together with its model or PoC implementation and each aspect finishes with a section of suggested improvements. Wherever the model section is left out this means that there was no model implementation of this aspect. If the PoC implementation is left out, this means that it was the same as the model implementation.

4.2.1.1 Determining Consumer rate

Purpose To re-evaluate the number of consumers submitted every five seconds, according to the queue length history of the last minute the processing capacity is scaled up or down.

Model solution An upper limit and lower limit are defined to indicate the sufficiency of the processing capacity in terms of consumer rate. The last minute of history is used to determine the state in which the system currently resides. Going back from the most recent measurement, the number of successive upper limit breaches is counted; when the sequence stops the count is terminated. Also, the number of times the queue length was below lower limit is counted. This results in three possible outcomes. First, if there is too much processing capacity, indicated by the count of lower limit breaches. When the history was below the lower limit more than 4 times, the consumer insert rate is lowered with 10%. Second, if processing capacity proves to be insufficient, the consumer insert rate is raised with 20% for every time the upper limit was breached, with a maximum of 100% if all history items breached upper limit. In the third situation, neither of these events occur, in which case processing capacity is assumed to be sufficient. No changes are made in this situation.

This procedure allows fast adaptation to growing event insert rates, potentially doubling the consumer insert rate every minute, while taking gradual steps during scale down.

Improvements Ideally, the consumer insert rate could be re-evaluated using the event insert rate of the monitored container, because there is a direct relation between the number of needed consumers and the event insert rate. Using queue length to steer this process introduces errors, because a rise in the queue length could mean two things:

- Inserts/sec have risen and there is not enough processing capacity.
- The underlying system is overloaded and there is a backlog of consumers that need to be run.

In this algorithm the first reason is assumed and the second is handled by the built-in overload prevention, which is explained below.

The exact figure for the optimal upper limit is not stated here; it is however explored in the results section of the model (section 4.2.2.3 on page 44).

4.2.1.2 Overload prevention

Purpose Overload prevention is necessary to make sure that when the batch system is overloaded, no more consumers are submitted into the system. When overload occurs, the batch system is unable to process all submitted consumers. This will cause a rise in the queue length. The re-evaluation of the consumer rate described above will assume this is caused by a risen number of inserts and will raise (worst case: doubling each minute) the consumer rate, further overloading the system.

PoC solution Each manager keeps track of the number of consumers that are submitted to the batch system, but that still need to run. Using this number the manager can determine whether an overload is situation exists and stop submitting more workers.

Improvements The central storage needed for the count introduces a potential risk into the system in terms of availability. Therefore it would be better if the batch system could report, for each type, on the number of workers that are waiting to be run.

The only available method to monitor job progress in the current batch system is to read and parse the log files (provided by the batch system) for each job. The drawback of this is the potentially large number of log files that have to be monitored. This is why job tracking is implemented in the managers and workers.

4.2.1.3 Backlog

Purpose A backlog in the events that need to be processed can be caused by three things:

- an empty queue on startup
- a sudden burst of events for which processing capacity was insufficient
- a systematic shortage of processing capacity because either the batch system or mysql can not provide sufficient resources

Model solution To secure the processing of a backlog, the system regularly checks the queue length and after this exceeds a certain threshold it will submit extra workers. Because the cause can be overload of either the batch or database system, it is also subject to overload prevention.

Improvements The current implementation only helps to process events already waiting. It might be worthwhile to experiment with anticipating future events and starting more workers than necessary.

4.2.2 Model

This section explains the model implementation for the consumer/h-worker system according to its purpose, environment and results.

4.2.2.1 Purpose

The model was developed to create a sound core algorithm that can be used while building the PoC. Doing this in a model has several advantages:

- early check on its feasibility.
- Tests are much faster since the conception of time is modeled as well.

Upper limit	total waiting events	run workers
50	230269	16319
250	618564	10766
500	1093347	9461
1000	1449517	10175
2000	3047514	9982
5000	9448519	9614
15000	42139598	8375
current system best case	-	12000

Table 4.2: Total number of worker runs needed to process all events. The upper limit is changed in order to see the effect this has on the total worker runs and the number of measured events in the queue. The events have a maximum insert rate of 50 events/sec.

4.2.2.2 Environment

The model was written as a program in Python, which wrote data to files that could be generated into plots. This way the algorithm could be easily analyzed and adjusted. The model was tested with a sinus wave as its input rate, to mimic the changing input rate during the day in the live system. While this is not life like, it provided enough data to adjust the algorithm. Two maximum insert rates of 50 and 1000 events per second were tested to examine the adjustment of the system to low throughput and high throughput queues. For each of these input rates a variety of upper limit settings were tested, in order to determine their effects on the algorithm.

A worker in the system processes 50 events before it quits. In a later test exploring the PoC conditions, event insert rate was set a maximum of 3000 and the workers processed 400 events at once. This was done to limit the number of workers needed in the system.

4.2.2.3 Results

The first version of the algorithm did not include a system to cope with the accumulation of events while the algorithm adjusts its insert rate to match conditions on startup. The tests showed early on that this created an unstable system on startup. The worker insert rate over shot the ideal in order to process all waiting events. A backlog system remedied this by submitting workers to process the backlog that was created during abrupt changes in the event insert rate. For more information about this please see section 4.2.1.3 on the preceding page and section 4.2.3.3 on page 47.

As can be seen in the plots in section B.2.1 on page 89, the algorithm adjusts well to changing conditions. The upper limit used to determine whether the worker insert rate should increase is has a lot of influence on how efficiently the algorithm runs. There is a trade-off between the delay events incur before being processed and the total number of workers run.

Tables table 4.2, table 4.3 on the facing page show that when the the upper

Upper limit	total waiting events	run workers
50	7144436	211115
250	7452414	175680
500	9909419	167900
1000	10599948	158138
2000	12052485	157162
5000	16591509	155812
15000	32221041	147241
current system best case	-	240000

Table 4.3: Total number of worker runs needed to process all events. The upper limit is changed in order to see the effect this has on the total worker runs and the number of measured events in the queue. The events have a maximum insert rate of a 1000 events/sec.

Upper limit	total waiting events	run workers
50	18489993	97404
250	18711647	93843
500	19049233	81263
1000	20209587	74626
2000	23171925	70592
5000	30052367	62926
15000	43745289	60734
current system best case	-	90000

Table 4.4: Total number of worker runs needed to process all events. The upper limit is changed in order to see the effect this has on the total worker runs and the number of measured events in the queue. The events have a maximum insert rate of 3000 events/sec. Each worker processes 400 events on a run

limit used in the algorithm is increased, the total number of workers decreases but the number of events that are measured to be waiting increases, indicating that events incur more delay before being processed. The tables also show the "current system best case", which the number of consumers that would need to run in the current system to match the peak insert rate.

A sweet spot appears to exist when the upper limit is set between 250 and 2000 for a worker that processes 50 events in a run. This changes to 500 up to 5000 when the worker processing capacity is changed into 400 events.

Further tests regarding the optimal settings for the algorithm are left for future work. It is however proven that, at least in the model, the algorithm uses less resources than the best case scenario in the current system. For example, a maximum insert rate of a 1000 events per second and a worker processing capacity of 50 events leads to a static configuration of 20 h-workers that need to run each second. This results in 24.000 h-worker runs during the period of the model test. Table 4.3 shows however that the algorithm used 30 to 40% less workers.

In conclusion, the algorithm adjusts very well to changing conditions in the insert rate which is one of the main demands. By design, demands like failure

resistance and single instance daemon are delegated to the batch system. This only leaves the ability of a manager to restart without seriously hampering the processing capacity, which is the purpose of the backlog aspect. As said the optimal settings for the algorithm can be even further explored, but this is designated to be of lesser importance and as such left for future work.

4.2.3 Proof of Concept implementation

This section explains the PoC implementation for the consumer/h-worker system according to its purpose, environment and results.

4.2.3.1 Purpose

This system is the most lifelike test in the research. The amount of factors taken into account, make this phase more realistic than the model. This results in a test system which can mimic the live behavior as closely as needed in order to evaluate the system.

This evaluation takes places using the focus points or problems that exist in the current system.

4.2.3.2 Environment

The queue system submits managers to the batch system when a new container is created. The batch system then runs these managers and the workers that they in turn submit into the batch system. The batch system consists of a total of 11 nodes; three of these are the nodes that run the queue system which can only submit managers to be run on the other eight nodes. Each node has a number of slots, in which they can execute a job. In total the system has 68 slots.

The chosen batch system is Condor and this choice is explained in section A.1 on page 55.

The input data for each manager is dependent on the insert rate for its associated container. Still, the quality of the event stream (see section 4.1.3.2 on page 38) is such that it closely follows what happened on the alert queue for a typical day in april 2008, making sure that the pattern represented by the stream is as realistic as possible. However, in order to generate enough load the insert events are multiplied around 400 times.

While the queue managers are implemented as if this were the live system, the workers take a more simplistic approach. To save on the needed processing capacity in the PoC, these workers just delete X events after which they sleep for a short period to emulate processing the deleted events. Another reason for this simplistic approach is that there was no time to build a complete live test environment in order to give the workers proper access to all databases. Other aspects that would potentially make it time consuming are: the code is unfamiliar and the practice of ruling out any effect on the live or development environment would have been a trial and error method. The aim was to eliminate as many external factors as possible, making sure that they could not influence the test results.

4.2.3.3 Results

To evaluate the system, it is best divided into sections which will be discussed separately. The order in which they are presented here indicates their importance to the final conclusion.

Scalability In order to fulfill the scalability requirement the system is expected to scale according to needs. In case of the manager/worker model, the worker insert rate should follow a pattern, equal to the event insert rate on the associated container, thereby adjusting processing capacity to the requirements at hand.

The plots in section B.2.2 on page 111 show that this is indeed the case. Each plot illustrates the behavior of a manager associated with a container. There are five distinct container types and three nodes in the queue system, resulting in 15 possible containers and thus 15 plots. The plots contain four graphs; the first graph shows the estimated event insert rate of the queue, the weight of the container and the derived insert rate on that container. The second plot shows the worker insert rate determined by the manager. The third plot shows the number of waiting events (queue length) measured in the system. When no waiting events were measured this is not shown. The fourth graph shows an approximation of the processing delay over time. The reliability of this has a direct relation with the event insert rate on the container, and fluctuates strongly when the insert rate draws near zero. Nota bene: the event insert rates are unknown in the live environment.

By comparing figure B.43 on page 112 and figure B.48 on page 117 in particular it is seen that the system works well for both high capacity and low capacity queues.

However when a queue node becomes overloaded, the managers associated with a container on that node further degrade performance. This is shown by the plots in section B.2.3.1 on page 127. The node on which the container exists becomes overloaded around 23:45 (figure B.59 on page 129). This results in degraded performance from the database. Queue operations take longer and might eventually time out. Because queue operations take more time, connections stay open longer, further degrading performance of the already busy database system. In addition the manager detects a large number of waiting events and raises the worker insert rate even further, which can be seen in figure B.58 on page 128.

In addition the workers take longer to run and could thus fill up the resources of the batch system, thereby decreasing performance for all other jobs running on the batch system. This in turn could raise the number of waiting events in other queues, triggering their managers to raise the worker insert rate. Future versions of the manager algorithm should take the state of the node on which their container resides into account, for example by not further raising the worker insert rate when the node state is TOOBUSY. Another method of overload detection could be to monitor the rate at which workers are executed.

To allow the manager to adjust to abrupt and large increases in the event insert rate on a container the backlog system is used. As figure B.61 on page 132 shows, the backlog system submits burst workers when a backlog of events is created. These workers are to process the events currently waiting. This gives the algorithm time to adjust its worker insert rate to the desired level. This

can be seen by comparing figure B.61 on page 132 and figure B.62 on page 133. In B.62 the backlog system is disabled. It shows that the worker insert rate then overshoots the desired insert rate by a factor of two.

Failover Failover of the manager/worker system is handled by the batch system. For this, it is necessary that the managers are stateless, so that, in case of a crash, they can be restarted without undue problems. Of course, it is to be expected that processing capacity will be disrupted for a period of minutes after a crash. When a manager stops running the batch system will restart the manager on another node. figure B.61 on page 132 shows that, at 13:10, a manager crashes and restarts almost immediately, after which the necessary processing capacity is restored within minutes.

Single instance daemons This specific kind of worker can only be run once at any time, to avoid data corruption. The managers from the manager/worker system are implemented as single instance daemons and therefore serve as perfect examples.

Modification of existing code One of the goals of the research is to use the legacy system as a basis for a new solution. This should be done by re-using codes and techniques that are already implemented, but only to a certain extent, as the objective is to develop a new and improved method. All consumers and h-workers in the current system must be rewritten to take configuration variables regarding the location of their containers/queues from either command line arguments or environment variables. They must also be stripped from the outer batch loop that made them run perpetually. The managers have to be developed, but can borrow heavily from the version built for this proof of concept.

Automation The manager/worker system proves to automatically scale processing capacity according to the needs.

Efficiency Adjusting the number of workers to the demand, should lead to more efficient use of hardware. The model tests of the manager/worker system show that this is indeed possible. A quick glance at the plots of this PoC test show that this is not yet the case, this is expected to be solved by revising or fine-tuning the algorithm at a later stage, when the input of the system is less focused on stress testing and becomes more realistic.

Prioritization at overload One of the subgoals of this research is to introduce prioritization in the system to handle overload cases as efficient as possible. In such cases, it is desired that jobs get an order of importance and resources are assigned to them accordingly. Because prioritization is defined to be of limited importance and time was short, this was not implemented in the PoC. Implementation could be achieved using the prioritization facilities provided by the batch system[2].

[2] `http://www.cs.wisc.edu/condor/manual/v7.0/2_6Managing_Job.html#sec:job-prio`

Monitoring Monitoring has a double purpose. On one hand it keeps track of performance indicators, thereby showing the current state of the system. On the other, it informs the team in case of errors. The possibilities to extract enough performance indicators is illustrated by the abundance of data collected in the PoC solution. In a life solution, all this data can be gathered and presented by the systems currently in place: Ganglia and Nagios.

Software/hardware demands of jobs In a live environment, it is not uncommon that different pieces of software have different requirements regarding software libraries, or maybe even hardware aspects. In the used batch system, such requirements can be taken into account during resource assignment. This is done by having execution nodes broadcast their capabilities and defining requirements for a job on submit. The batch system will then run the job only on the nodes that meet the requirements. For examples see the manual[3].

Minimizing network overhead In a later stage, the queue system and the batch system can possibly run on the same set of nodes. This would be done by dividing local resources between both system, for example: 30% for the queues and the rest for the batch system.

On submitting workers, a preference for a specific node can be defined. By defining this preference[4] for the node that holds the container, the job basically requests to be run close to the queue so that it can use unix sockets instead of tcp/ip. But when the node is busy the job is run on another node, so network overhead is minimized without making the worker wait. Configuring[5] the 'shared' nodes such that they only allow jobs that have requested to run on that particular node, makes sure that these nodes are only used by the jobs that take most benefit.

Load balancing Managers submit their workers to the batch system, which handles their execution. The batch system distributes them among the available nodes, making sure that the load is equally spread.

4.2.4 Conclusion

As described in previous sections and can be seen in the plots (section B.2.2 on page 111) the system works well in most respects. It adjusts well to changing conditions, although efficiency could be improved as is shown in the model. Failure resistance is robust and a crash of the manager is remedied within minutes, with only a small hiccup in processing capacity.

Weak spots however are overloads of the node on which the associated container resides or an overload of the batch system, for which two solutions are suggested.

The principle of using a manager/worker system in combination with a batch system (resource manager) is proven to be sound and effective and as such

[3] http://www.cs.wisc.edu/condor/manual/v7.0/2_5Submitting_Job.html#SECTION00352000000000000000

[4] http://www.cs.wisc.edu/condor/manual/v7.0/2_5Submitting_Job.html#SECTION00352100000000000000

[5] http://www.ccp4.ac.uk/ronan/condor_tutorials/scotland-admin-tutorial-2004-10-12.html#start_expression

also applicable for the offloaded tasks. The only relevant question remaining is whether the batch system can fulfill the resource requirements of these offloaded tasks, no problems are foreseen though.

Chapter 5

Conclusion

It was a challenge to find a way to improve the current static implementation of h-workers and consumers without changing it in an intrusive way. While a lot of work still needs to be done before an actual implementation can be realized, this thesis presented a framework that increases the scalability, fail over and load balancing characteristics of the current offload and data preparation tasks.

The outlined solution facilitates the dynamic assignment of queues and tasks according to needs and availability of resources, introducing flexibility into the system. This flexibility is harnessed to improve on many of the deficiencies of the current system.

The static configuration of the system, mentioned as one of the core problems, is now done dynamically. In this system each task has a manager, responsible for requesting adequate resources, and the underlying batch system then assigns resources on one of the available servers.

The other core problem was that performance data per task is not available. By introducing the managers there is now a central authority per task that can gather and report performance data, helping in the analysis of problems encountered during daily operations.

These improvements allow the system to become more scalable because the batch system handles the physical resource assignment in an automated and dynamic way. Meanwhile the managers allow tasks to scale according to their individual capacity needs.

Fail-over of tasks is achieved by ensuring that the batch system restarts crashed managers. The algorithm in the managers must be designed to easily recover from a crash.

The tasks managers request resources to start short running workers. The fact that these workers are relatively short lived aids the fail-over and load-balancing characteristics. The fast pace at which workers are started allows the batch system to "balance" the workers over the available resources and quickly adapt to changing conditions.

In opposition to the chosen solution one might say that the design is too complex. However this complexity is necessary, because the system is designed to grow with the capacity needs of the coming years and the dynamic assignment of resources opens the doors to using unused resources on servers during off-peak hours.

There are several options left to improve the queue system, like: more ele-

gant handling of overload situations. A more suitable algorithm of the database queue, and the actual database product used are also variables that could provide further research.

Chapter 6

Future work

Throughout this thesis several possibilities for future work are mentioned. A summary of them is given here.

First, the algorithms used throughout the system could be further developed, thereby taking into account aspects like history. For example a heuristic container selection or worker insert rates that are partially based on history. It would also be interesting to see whether moving a small percentage of all containers on a node would be more effective, especially in combination with a back-off period in the queue manager in case of a TOOBUSY node.

Better results however, could probably be achieved by finding a more suitable implementation for the queues. Overload detection in the queue managers could possibly be made more effective by monitoring the number of run workers per second.

The different settings in the queue manager algorithm should be explored in order to improve efficiency.

Other areas of research could be the development more effective resource management for SMP machines, instead of the current slot mechanism. Because many similar jobs execute in this system, their resource usage could be profiled over several runs. This profile could then be used to decide whether a node has enough spare capacity to run a job.

The Hyves serverpark and software should be explored for tasks that would benefit from running on a batch system, such as performing backups or parsing log files.

Appendix A

Complimentary explanations and data

A.1 Selection of a batch system

The selection of the batch system is based on a number of factors, ranging from failure resistance to costs. Three batch systems were tested on all relevant aspects; a small test setup was created to evaluate the operational difficulty of each system. All three batch systems will be listed below and for each all aspects will be discussed.

A.1.1 Sun grid engine

Sun grid engine (SGE) is an open source project sponsored by Sun Microsystems. The system works by defining a queue for each job category. A category is the product of several attributes like priority, software requirements, timing, etc. A queue can be associated with one or more nodes, and a node can, in turn, be associated with one or more queues.

As the batch system will be used in a facilitating manner, it has to be dependent. In other words, it needs to run perpetually. SGE works with a master that directs all other nodes. In case of failure the system stops working. To eliminate this risk, shadow masters can be designated to take over the role in case of master failure. In order to facilitate this, a shared file system is required to share state between the master and its shadows. Fail over should happen within ten minutes.

By assigning a queue only to machines with a certain software package, it is possible to make sure that jobs that need this package will only run on these nodes.

Depending on the policy jobs that have failed to execute properly can be re inserted into the queue, so that they will be run again.

As mentioned, queues can be assigned a priority, thereby influencing priorities among jobs.

Queues can be suspended according to a schedule. Jobs in the queue are not brought into the system during its suspension and jobs already running are suspended. Jobs can thus be started at a specific time, namely the time the queue is unsuspended, but the queue must remain unsuspended as long as a job runs. Some jobs are not allowed to leave the queue and are thus forced to

run at intervals. However while the queue is unsuspended the job will restart as soon as it finishes. For a well working system, suspension of the queue and the finishing of the jobs must coincide. If the schedule suspends the queue too early, the job is suspended as well and will not finish until the queue is resumed. If the schedule is too lax the job will be restarted as soon as it has finished.

A job inserted into the batch system is only run once at any time, however nothing stands in the way of inserting the same job multiple times. Consumable resources can be used to remedy this problem; they are often used for example for licensing. When a job indicates that it uses a consumable resource that resource is lowered as soon as it is run. When the resource only holds one (for example license), no other jobs that use that resource can run at the same time.

A.1.2 Condor

Condor is developed by the University of Wisconsin Madison. The system works by associating a set of attributes to a job, called an advertisement. Each node also advertises its capabilities. The scheduler then matches jobs onto the available resources using these advertisements.

Condor is a largely distributed system, but one of its few centralized aspects is the negotiator. The negotiator fulfills an important role by mapping jobs to available resources. The negotiator is also called the central manager. To protect the system against failure, several mechanisms are implemented. The most important of these protects the negotiator. From the condor manual[1]:

"Configuration allows one of multiple machines within the pool to function as the central manager. The other potential central managers are the idle central managers. Each potential central manager machine runs the high availability daemon, condor_had. These daemons communicate with each other, constantly monitoring the pool to ensure that one active central manager is available. If the active central manager machine crashes or is shut down, these daemons detect the failure, and they agree on which of the idle central managers is to become the active one.

In the case of a network partition, idle condor_had daemons within each partition detect (by the lack of communication) a partitioning, and then use the protocol to choose an active central manager. As long as the partition remains, and an idle central manager exists within the partition, there will be one active central manager within each partition. When the network is repaired, the protocol returns to having one central manager."

Replication from the active central manager to the idle central managers makes sure that the state is not lost during failure.

Another potential failure point is that each machine holds a queue of the jobs which are submitted on that machine. When that queue becomes unavailable, those jobs can no longer be run until the problem is fixed. A central queue (with backups) can be used to remedy this problem, but this method works through a shared file system.

[1] `http://www.cs.wisc.edu/condor/manual/v7.0/3_10High_Availability.html`

A job can specify requirements which must be fulfilled by the node that is matched to the job. These requirements can range from a software package to hardware requirements or free memory.

Depending on the policy in place, the job will be re inserted when it exists abnormally.

Jobs can be given a priority on submission, which is then taken into account by the negotiator.

Each job can also be given a schedule, which specifies when the job has to run. The job is then released on the times that are specified, unless it is still running from the previous interval. When a job finishes it is not run until the schedule releases it again.

Single instance daemons are implemented using a schedule that will run them every five minutes. In case of a failure it will take five minutes before the daemon runs again.

A.1.3 Cluster resources

Cluster resources is the company that builds Moab, a scheduler that is run on top of the open source Torque resource manager.

Moab relies on replication to provide high availability of its scheduler. The underlying resource manager (torque) however uses a shared file system to provide high availability.

Just as in the previous batch systems, jobs can specify requirements which have to be met when scheduled. Also the policy decides whether jobs are re inserted on failure.

Using a reservation system, jobs can be scheduled regularly or at particular times. The job is started each time, regardless of whether the previous job still runs.

By using consumable resources the system can guarantee that single instance daemons really only run once, even when submitted multiple times.

A.1.4 Conclusion and selection

Even though Condor is a complex system to operate, it best fulfills the aspects discussed in the previous sections. Failure resistance of the master is more than acceptable, although the queues could use more attention. Its crontab like scheduling works well with the way tasks manager would be constructed for workers (section 3.5.2.1 on page 28), and its priority and requirements mechanisms for jobs seem more simple than those of SGE. For these reasons Condor was chosen for a Proof of Concept implementation. For a real world implementation this decision should be reconsidered, in the view of knowledge gathered during this research.

A.2 Tasks implemented using h-workers

This section gives an indication of the tasks that are implemented using the h-worker paradigm. All tasks listed here should work in the new solution. The list does not aim to be complete, only to be comprehensive. The table shows the task names, the interval at which the task is run and whether it is a single instance.

Task name	Interval	Single instance
Easymembersearchindexer	daily	
AlbumIntegrityChecker	daily	
MemberIterator	daily	
SearchBuilder	daily	Yes
Globalcache	almost perpetual	
OrderIntegrityChecker	hourly	
OldMessageDeleter	hourly	
ServerManagementImporter	almost perpetual	Yes
Parsemail	almost perpetual	
WhatsNext	almost perpetual	Yes

Table A.1: List of tasks using the h-worker model

A.3 Tasks implemented using the producer/consumer paradigm

This section gives an indication of the tasks that are implemented using the producer/consumer paradigm. All tasks listed here should work in the new solution. The list does not aim to complete, only to be comprehensive. the table will mention task names and whether the task currently uses a distributed queue.

Task name	Distributed Queue
Queueserver	
SmsQueueServer	
OrderQueueServer	
QueueServerBlogPing	
QueueServerGroupMessage	
QueueServerOneWayFriendMessage	
FeedleecherQueueServer	
ProcessPrepaymentsDaemon	
ResendInvitationsDaemon	
MemberDeletion	
NewsLetterSender	
MailNotifier	yes
SignalQueueServer	yes

Table A.2: Dispersion of jobs on character categories

A.4 Database

A simple definition of a database is: A data store that allows efficient storage and retrieval of data. Several techniques exist that define different methods of storing and retrieving the data, some of these are object, relational and xml databases.

Relational databases are in common use among web applications. The MySQL database, `http://www.mysql.com` is such a relational database. It is also in widespread use, and has always been used by Hyves. Hyves makes extensive use of the MySQL database for data storage and retrieval. The 500 database servers in use by Hyves, currently contain about 4TB of unique data. This data is however duplicated a great many times over these 500 servers to facilitate load balancing and fast retrieval of data.

Several techniques are used to meet the data access requirements imposed by the volume of the website. First, duplicates of the data are spread over more servers, so that they can be accessed in parallel. Write operations are sent to a 'master' server which keeps all duplicates consistent through replication, which makes sure that all duplicates execute the writes in the proper order. Read performance can be scaled almost linear with the number of duplicates. Write performance however is restricted, and can only scale up to the capabilities of the weakest server.

Second, the data is split into distinct sets. For example, market data and friend data can be split because there are no relations between the two. Each set gets its own servers to apply principle one if necessary. This is an easy method of reducing the writes per dataset by splitting it over multiple sets. This however only works if distinct sets of data can be identified within the total dataset.

Third, the data can be split according to some common identifier. For example the data for the users 1 through 50.000 are put onto server X, while the data for the next batch of users is put on server Y. If the the size of the resulting partitions is chosen carefully the system can scale almost endlessly, as the partitions can be kept small enough to fit easily within the capabilities

of a database server. The problem has now shifted to determining which server holds the data we seek. More about these techniques can be found here: [11]

A.5 Load balancer

The load balancer used is a layer 4 load balancer, which means that it works on the ip level in tcp or udp. Clients connect to the load balancer and are then redirected to a real node chosen by the load balancer.

This behaviour is achieved by using direct routing. The load balancer rewrites the destination mac address and puts the frame on the internal lan where it is received by the real node. More about this and other possible methods to load-balance using layer 4 can be found in [39].

The connection scheduling algorithm used throughout this paper is Weighted Round Robin. A more detailed explanation of this and other connection scheduling algorithms can be found in [39], but here is a small explanation extracted from the article:

In the WRR scheduling, all servers with higher weights receive new connections first and get more connections than servers with lower weights. Servers with equal weights get an equal distribution of new connections. For example, the real servers A,B,C have the weights 4,3,2 respectively, then the scheduling sequence can be AABABCABC in a scheduling period ($mod \sum(Wi)$). The WRR is efficient to schedule requests, but it may still lead to dynamic load imbalance among the real servers if the request load varies highly.

A.6 Solution A

This design is based on , the article describes a load balancer algorithm in which each node in the system has the responsibility to manage its own resources. If the node becomes TOOBUSY it has to move work to other nodes.

Solution A is a distributed system in which each node is independent. Communication with other systems only happens in case of status updates or when moving work between nodes. The purpose of the system is to load-balance and scale typed containers and hardware. The number of nodes used by the system is scaled according to the needed capacity: all nodes during peak hours, only a few during off hours.

A.6.1 Node

A node is a hardware system that is available for use by solution A. Unlike nodes in 'Solution B' (section 3.5 on page 24) they will be used by queues, consumers and h-workers. Each node has a state indicating its availability. This state is determined by the total resource consumption on the node. Available states are as follows:

IDLE has no containers, is not used by the system

NORMAL has a normal workload

BUSY node is busy, and load balancing can not move work to this node

TOOBUSY node is overloaded and load balancing tries to move work to normal or idle nodes.

At a certain interval, each node takes a set of local decisions. These are based on local information and the information concerning all other nodes available through the global state. These decisions are further described in the sections section A.6.4 on the next page to section A.6.7 on page 63.

A.6.2 Container

Unlike in 'solution B', a container holds a fixed set of consumers or h-workers, all performing the same task. The h-workers and consumers run perpetually, just like in the current system. If a container holds consumers it also holds the associated queue (see figure A.1). Typing occurs according to the task the consumers or h-workers perform. Also, multiple containers can co-exist on the same node.

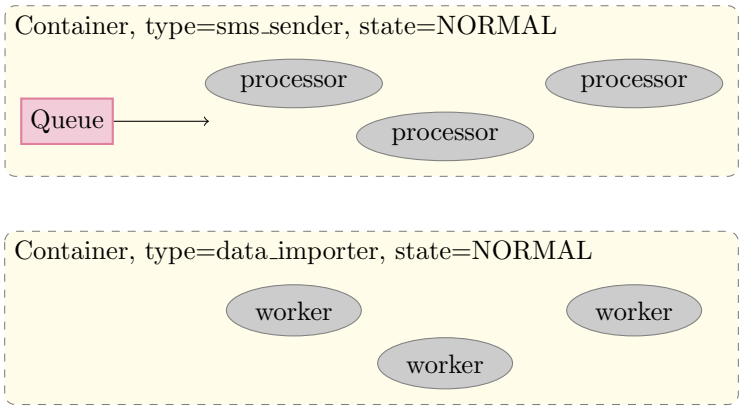

Figure A.1: Different container implementations

Because the set of consumers or h-workers running in the container is fixed, so is the maximum processing capacity per container. To facilitate scaling, each container therefore indicates, through a state variable, whether more processing capacity is needed. In such situations the system tries to start more containers of the required type in order to increase the processing capacity. 'consumer containers' might use the length of their queues to detect if more capacity is needed. A growing queue length generally indicates a shortage of processors. The 'h-worker container' has no real conception of how many h-workers are required, but simple first versions might try to aim for a number of h-workers and use the container state to reach this 'optimum'.

The queues in the system are segmented and a segment holding a container of the right type runs on each node. All incoming events are distributed among the available containers, taking into account the correct type. When a 'consumer container' runs on a node the local database will hold its queue-segment. Consumers therefore always connect through unix sockets, thereby minimizing network overhead. When multiple 'consumer containers' run on the same node, they will share the local queue-segment.

Each container has a start and stop method that is called by the system. This allows for different implementations per container type and easy adoptation to the different requirements per type. For example: a 'consumer container' might check whether a queue is available on the node before starting the consumers.

A.6.3 Global state

Every node in the system regurarly sends an update to every other node in the system announcing its own state: node name, node state and a list of containers. Every node processes the received updates, and is thus informed about the global state of the system.

This information is used for making local decisions based on the global state of the system. Solution B implements the same global state and for further discussion of membership, implementation and scalability please read section 3.5.1.3 on page 25.

A.6.4 Load balancing

Each node takes a number of local decisions. One of them is whether containers should be moved to other nodes. When the node is TOOBUSY, the algorithm will select a number of containers and attempt to move them to different nodes. NORMAL nodes already containing the container type are preferred, if these are not available the system uses other NORMAL nodes. When too few normal nodes are available, the target node is selected from the IDLE nodes. This is further explained in section A.6.5.

On movement of a container the system stops the container on the current node and restarts it on the target node. If this does not succeed the system leaves the container on the originating system. Stopping of a container immediatly drops the resource consumption on that node.

In order to distribute incoming events, a load balancer monitors the number of each container type on every node and distributes the events according to these numbers. The algorithm Weighted Round Robin is briefly discussed in section A.5 on page 60.

A.6.5 Scaling up/down

Both the number of active nodes and the number of containers per type need to be scaled up and down according to the required capacity.

Using the state variable of each container indicating whether more processing capacity is required, the percentage of containers (grouped by type) that request more capacity is calculated each decision cycle. When this exceeds a threshold the node will try to start more containers of the required type. These are started locally, if the current system load allows this. To make sure that not every node starts containers, only a percentage of the nodes that detect the need for more containers, succeed in starting a new container.

The same principle is used for determining whether more nodes are needed. When the majority of nodes is BUSY or TOOBUSY, the load balancing described above is influenced. An IDLE node is selected as a target node, effectively that node is taken into use. To make sure that not all nodes are taken

into use at the same time, only a percentage of the times an IDLE node is truly selected.

During off-hours the system is scaled down. When all nodes, or at least a large percentage, in the system indicate their states as NORMAL, there is a small chance that a node will try to move all of its containers to other nodes. If succesful the node returns to the IDLE state.

Also, during the off hours, a small percentage of the nodes that are not trying to reach the IDLE state will try to stop some of their containers. They will only stop containers that exist multiple times on the same node. This is in an attempt to consolidate the number of containers.

A.6.6 Failure resistance

There are several failures that can occur in such a system. The most common is the loss of a node. This could be because of communication failure, bugs in the software, hardware failure, etc. In such a case the node will no longer send periodic updates and will be removed from the global state. After a period of time, all other nodes will no longer know of its existence. The node is reintroduced into the system when the other nodes start receiving updates again. The load balancer also notes the unavailability of the node and will no longer send events to it. So, in case of failure the node is taken out of the system until it announces itself again.

To guarantee that a certain type of container is always present in the system it has to be running twice, each on a different node, so that it is still available when a node fails. If one container goes down with a node, it has to be restarted on another node to re-establish its failure resistance. Of course this course of action is not available for single instance workers, so the system must keep track of all single instance workers and if one fails restart it.

A.6.7 Monitoring

Because the queue system is to be developed in house, it can be made to work smoothly with the monitoring systems in place. An interface could be provided to pull status information from nodes or the system could push information to monitoring systems. Available information includes system resources used, the list of present containers and their states. Auxiliary intelligence must be built in to detect non running single instance workers or container types which should be in the system.

A.6.8 Single instance daemons

A single instance daemon is wrapped into a container which will never indicate a need for more capacity. The system will run only a single container of that type and upon detecting failure, will restart the container on another node.

A.6.9 Conclusion

The system is a good fit for consumers that have continuous resource needs. The h-workers however, which have a burst need for resources at specific intervals, are ill served by this system.

Also the efficiency of the used resources is not optimal. Downscaling only happens during off-hours and a task that runs in the morning will keep its requested resources until the system downscales in the night.

All in all, the scaling mechanisms used are an ill fit for the h-workers, making the system unsuitable.

Appendix B

Plots

This chapter contains the plots used to explain the results in proof of concept. a section is created to differentiate between plots and there sources.

B.1 Queuesystem

B.1.1 Model test run

This section contains the plots for the queue model test run, discussed in section 4.1.2 on page 36.

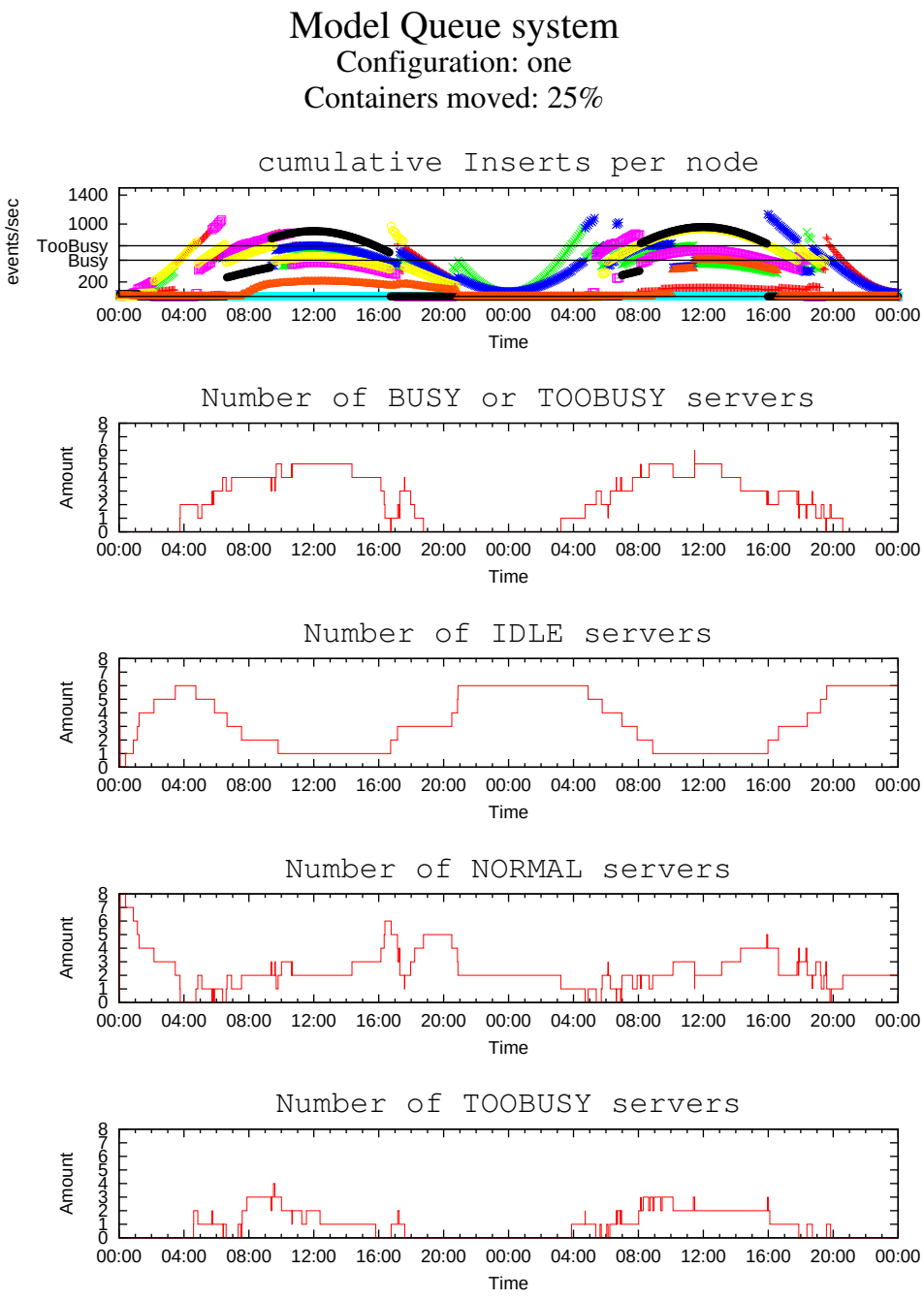

Figure B.1: Together with B.2, B.3 and B.4 this plot tests moving 25% of the containers in all four configurations. Plots B.5, B.6, B.7 and B.8 do the same, but move 40% of the containers.

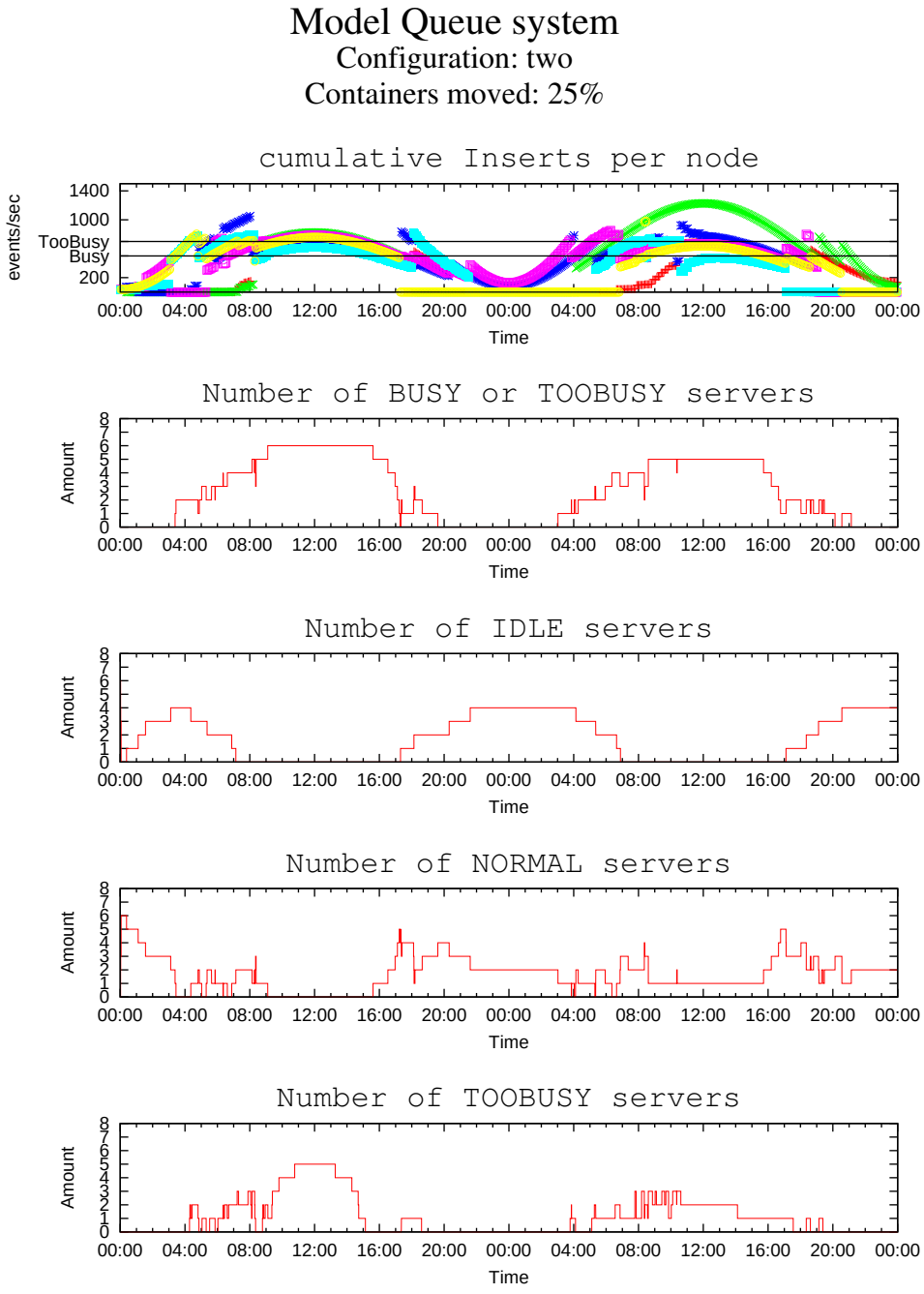

Figure B.2: Together with B.1, B.3 and B.4 this plot tests moving 25% of the containers in all four configurations. Plots B.5, B.6, B.7 and B.8 do the same, but move 40% of the containers.

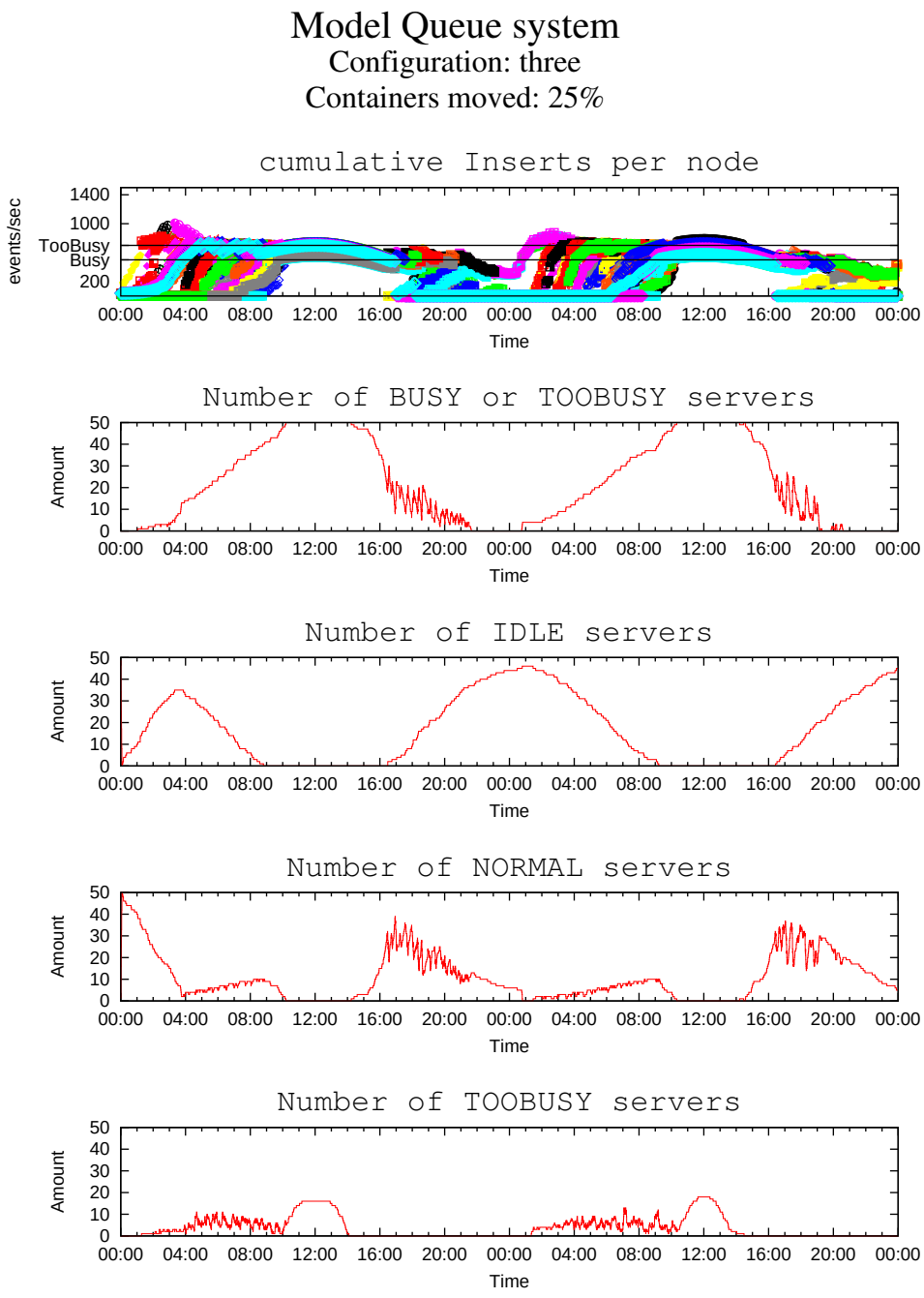

Figure B.3: Together with B.1, B.2 and B.4 this plot tests moving 25% of the containers in all four configurations. Plots B.5, B.6, B.7 and B.8 do the same, but move 40% of the containers.

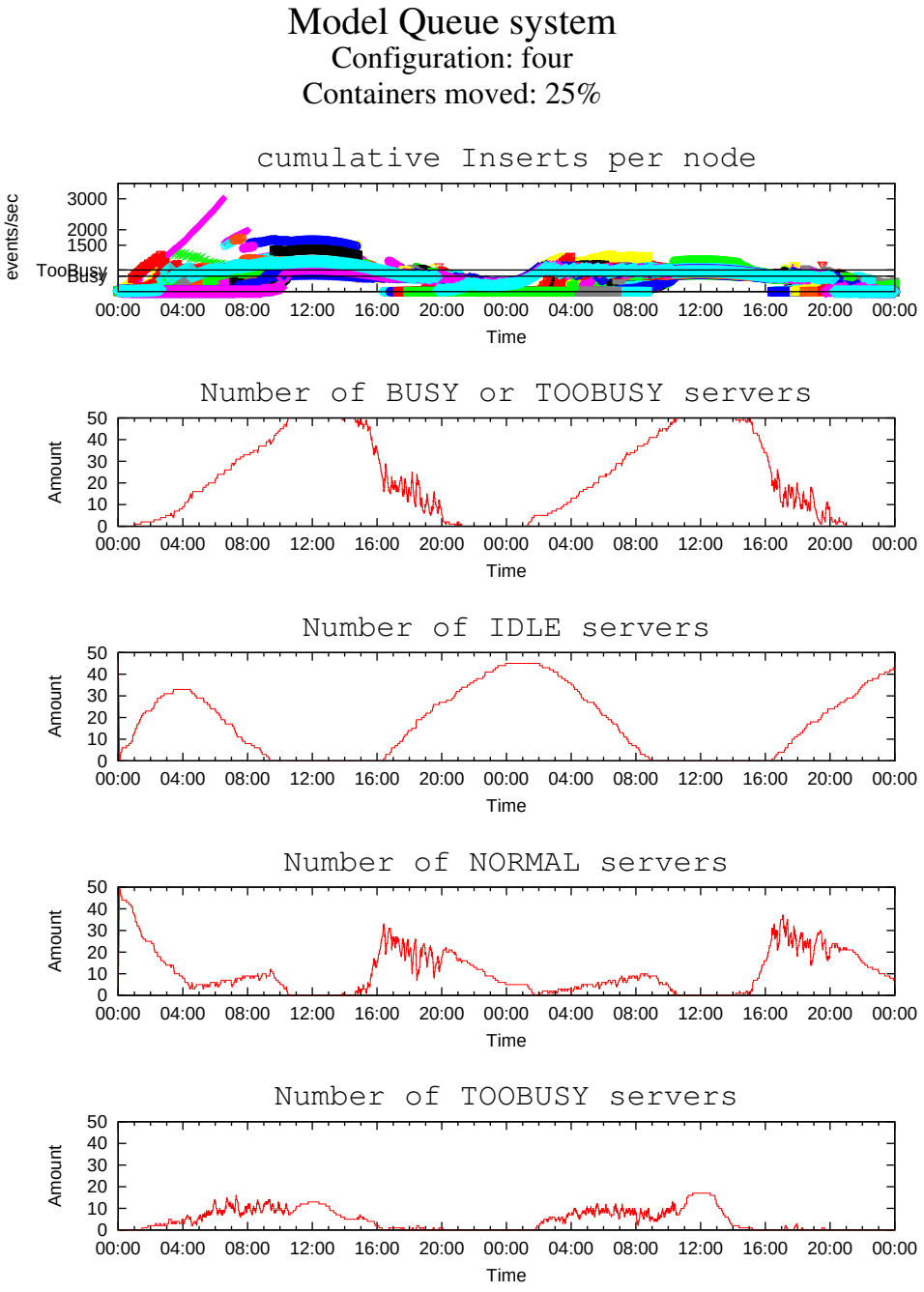

Figure B.4: Together with B.1, B.2 and B.3 this plot tests moving 25% of the containers in all four configurations. Plots B.5, B.6, B.7 and B.8 do the same, but move 40% of the containers.

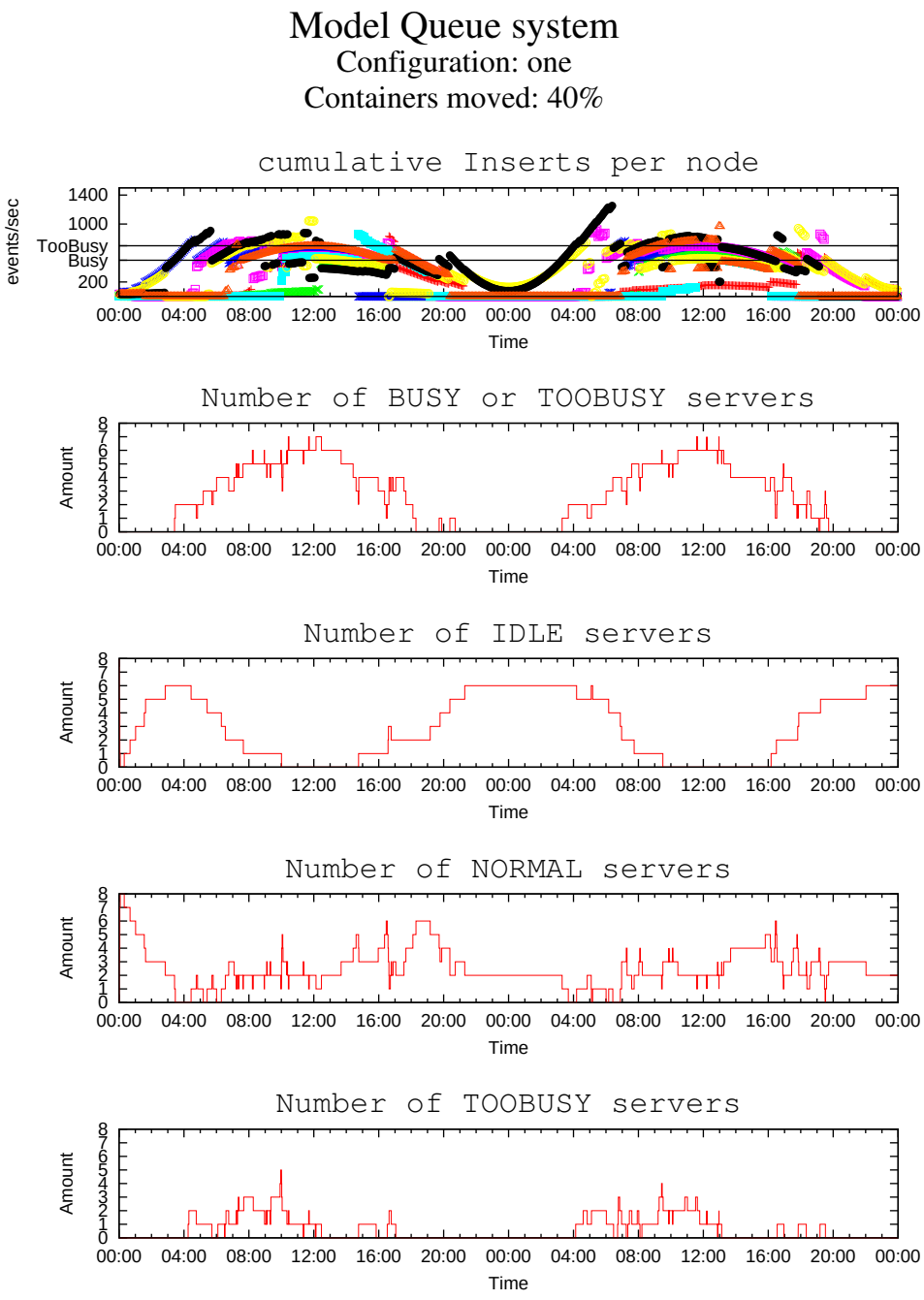

Figure B.5: Together with B.6, B.7 and B.8 this plot tests moving 40% of the containers in all four configurations. Plots B.1, B.2, B.3 and B.4 do the same, but move 25% of the containers.

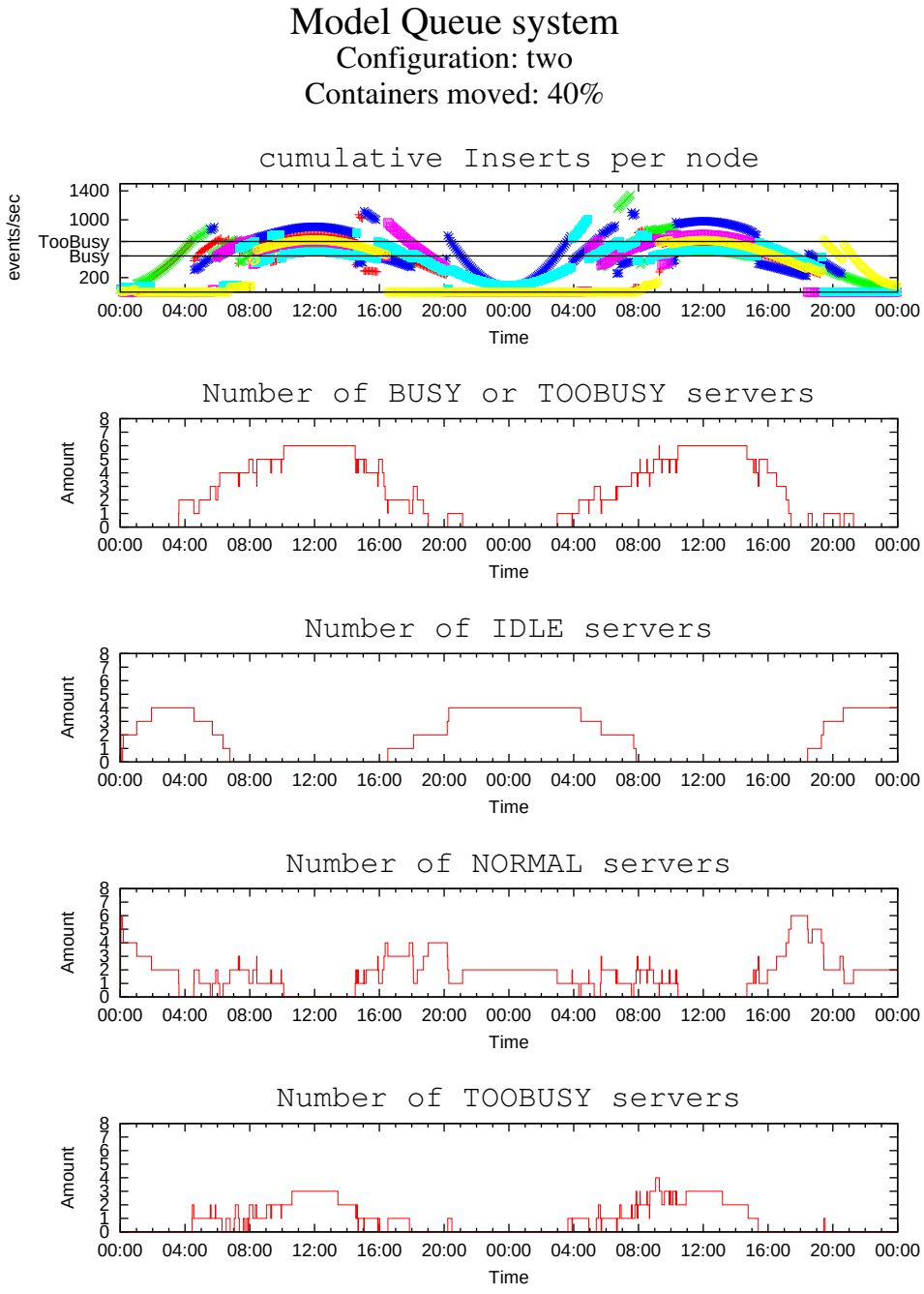

Figure B.6: Together with B.5, B.7 and B.8 this plot tests moving 40% of the containers in all four configurations. Plots B.1, B.2, B.3 and B.4 do the same, but move 25% of the containers.

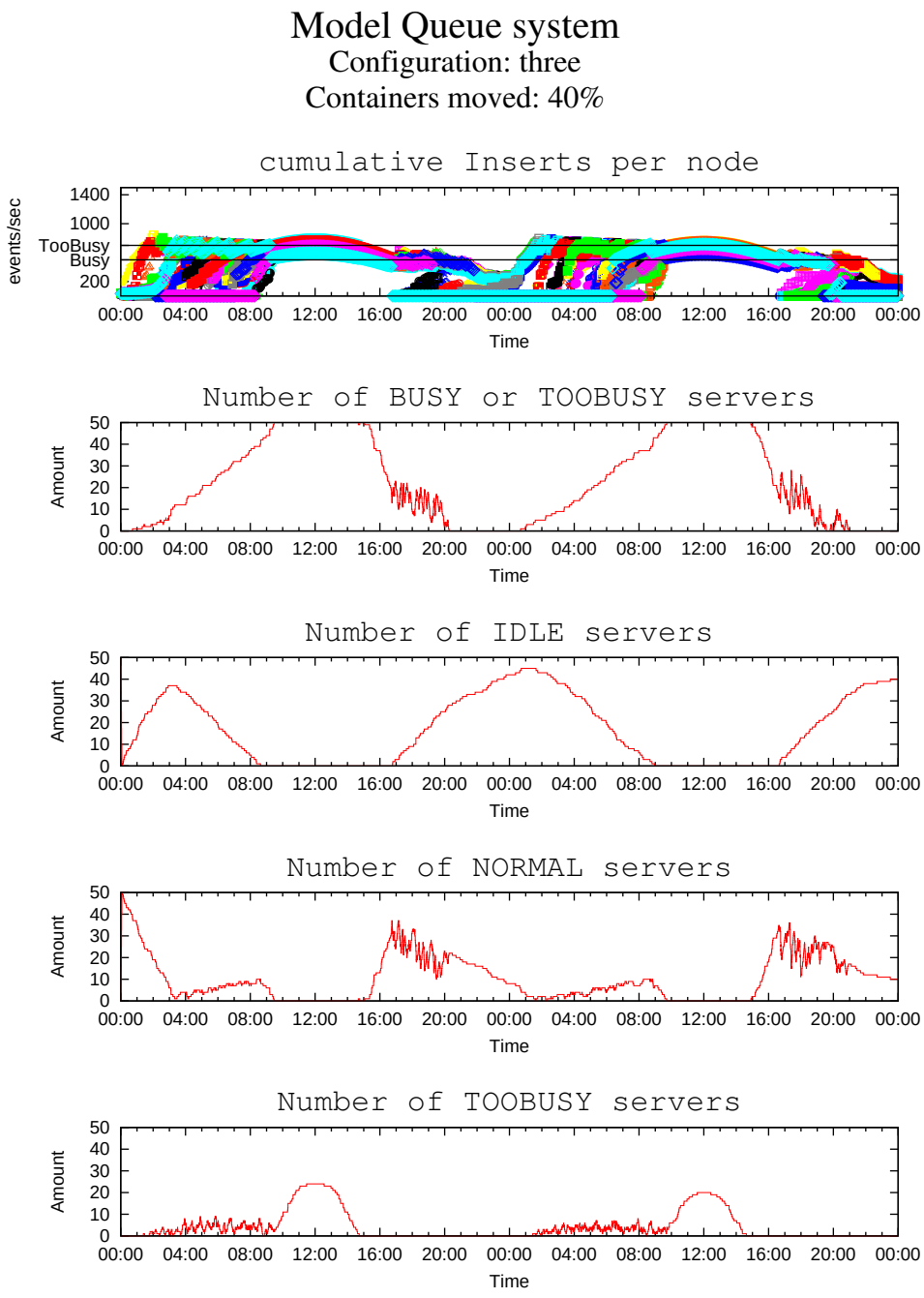

Figure B.7: Together with B.5, B.6 and B.8 this plot tests moving 40% of the containers in all four configurations. Plots B.1, B.2, B.3 and B.4 do the same, but move 25% of the containers.

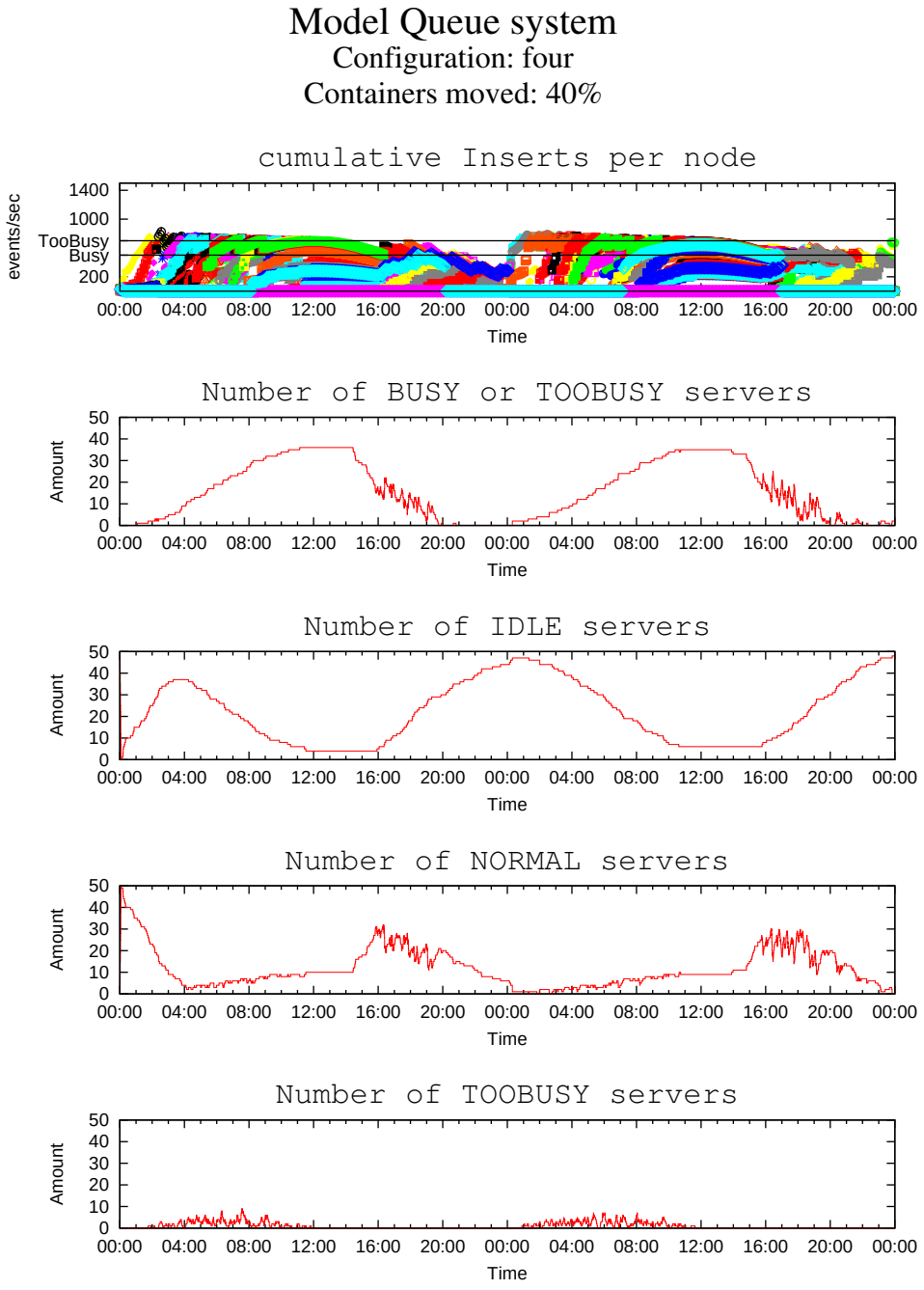

Figure B.8: Together with B.5, B.6 and B.7 this plot tests moving 40% of the containers in all four configurations. Plots B.1, B.2, B.3 and B.4 do the same, but move 25% of the containers.

B.1.2 PoC test run

This section contains the plots for the queue PoC test run, discussed in section 4.1.3 on page 38.

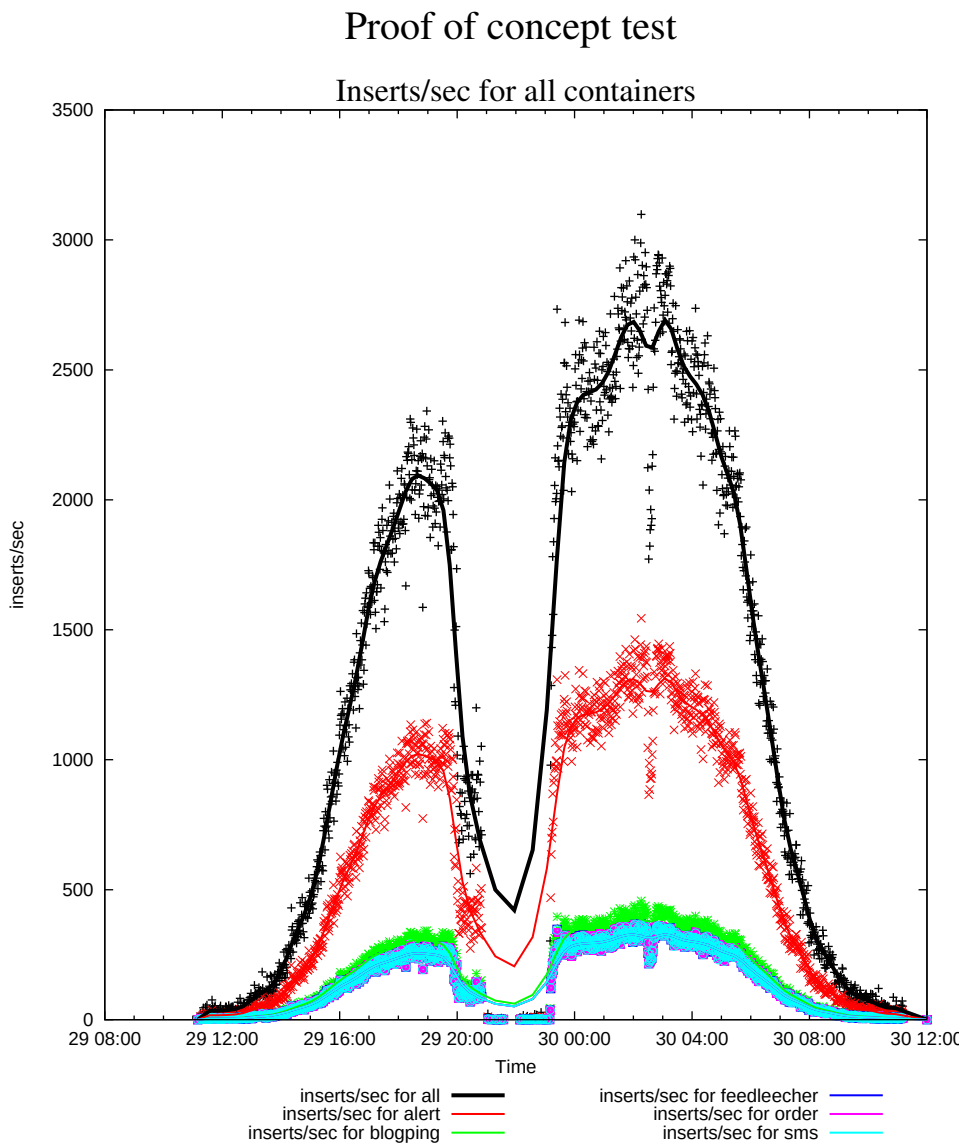

Figure B.9: Together with B.10, B.11, B.12 and B.13 this plot shows the state of the queue system. It shows the number of inserts per queue. These events are divided over the containers (B.10) and the containers in turn are distributed over the available nodes (B.11, B.12, B.13)

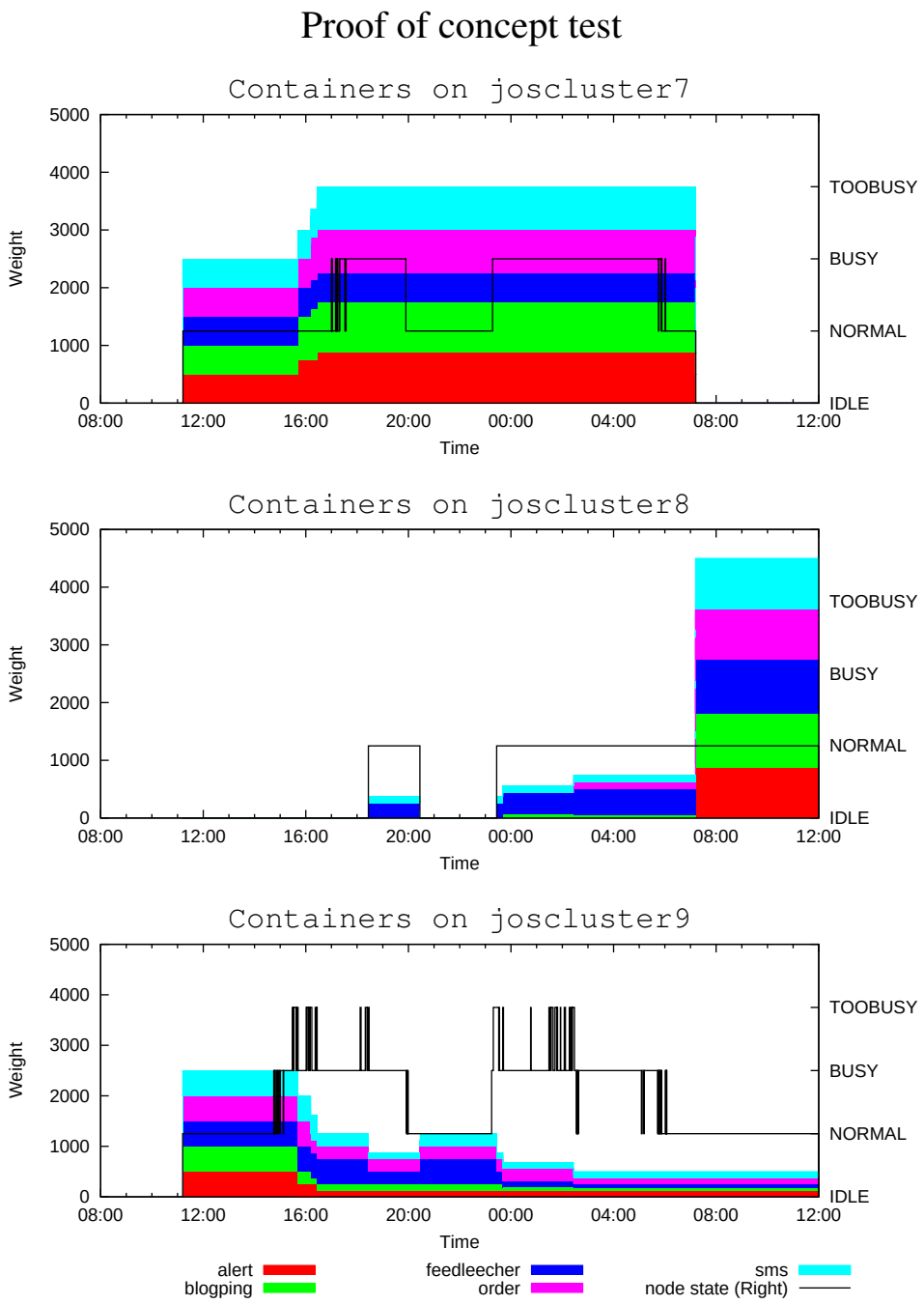

Figure B.10: Together with B.9, B.11, B.12 and B.13 this plot shows the state of the queue system. It shows the containers and there cumulated weight for each node over time. The events inserted according to B.9 are distributed over these containers and the containers in turn are distributed over the available nodes (B.11, B.12, B.13).

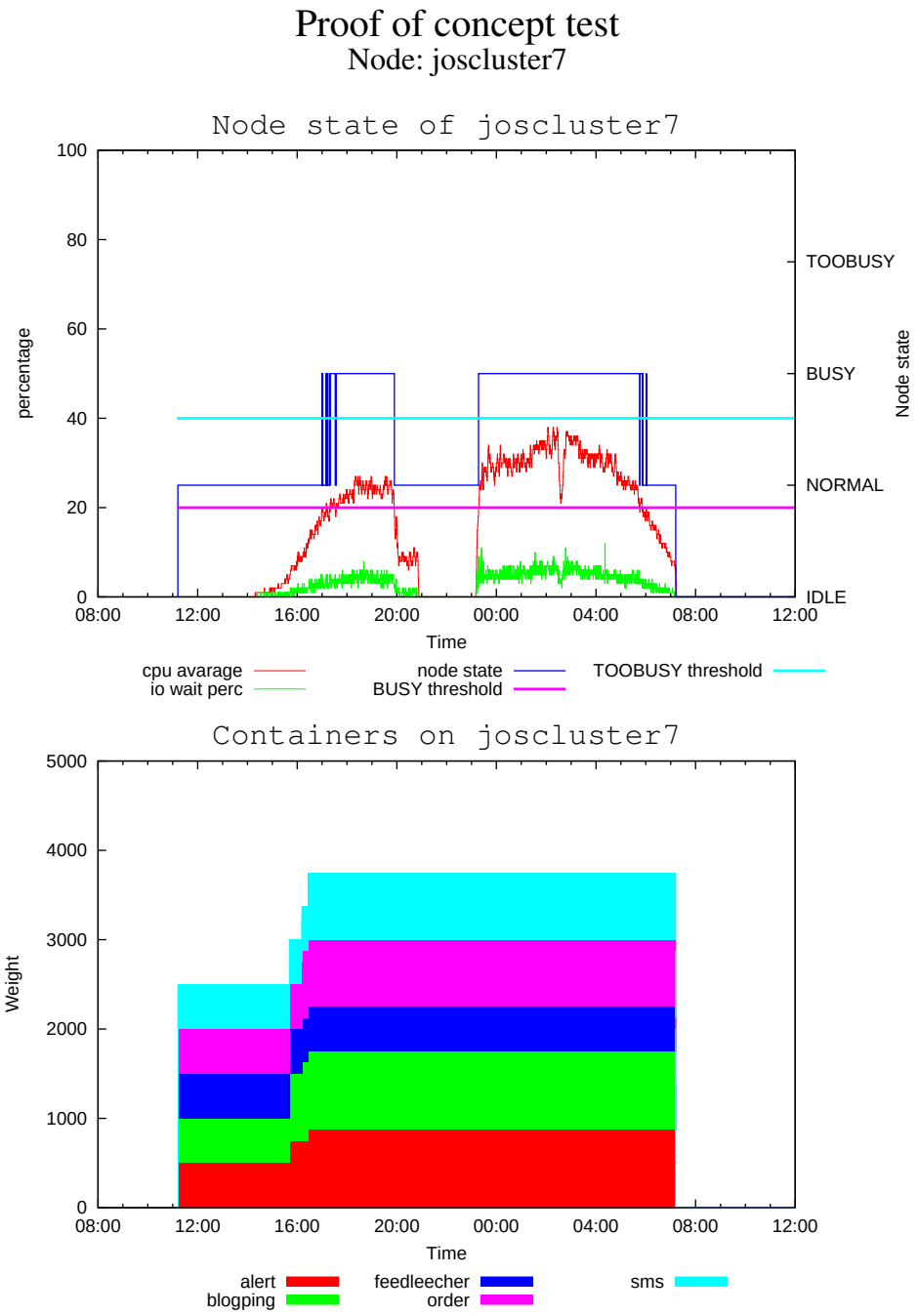

Figure B.11: Together with B.9, B.10, B.12 and B.13 this plot shows the state of the queue system. It shows cpu and node state in conjunction with the containers and there accumulated weight. The events inserted according to B.9 are distributed over the containers (B.10) and the containers in turn are distributed over the available nodes (B.11, B.12, B.13).

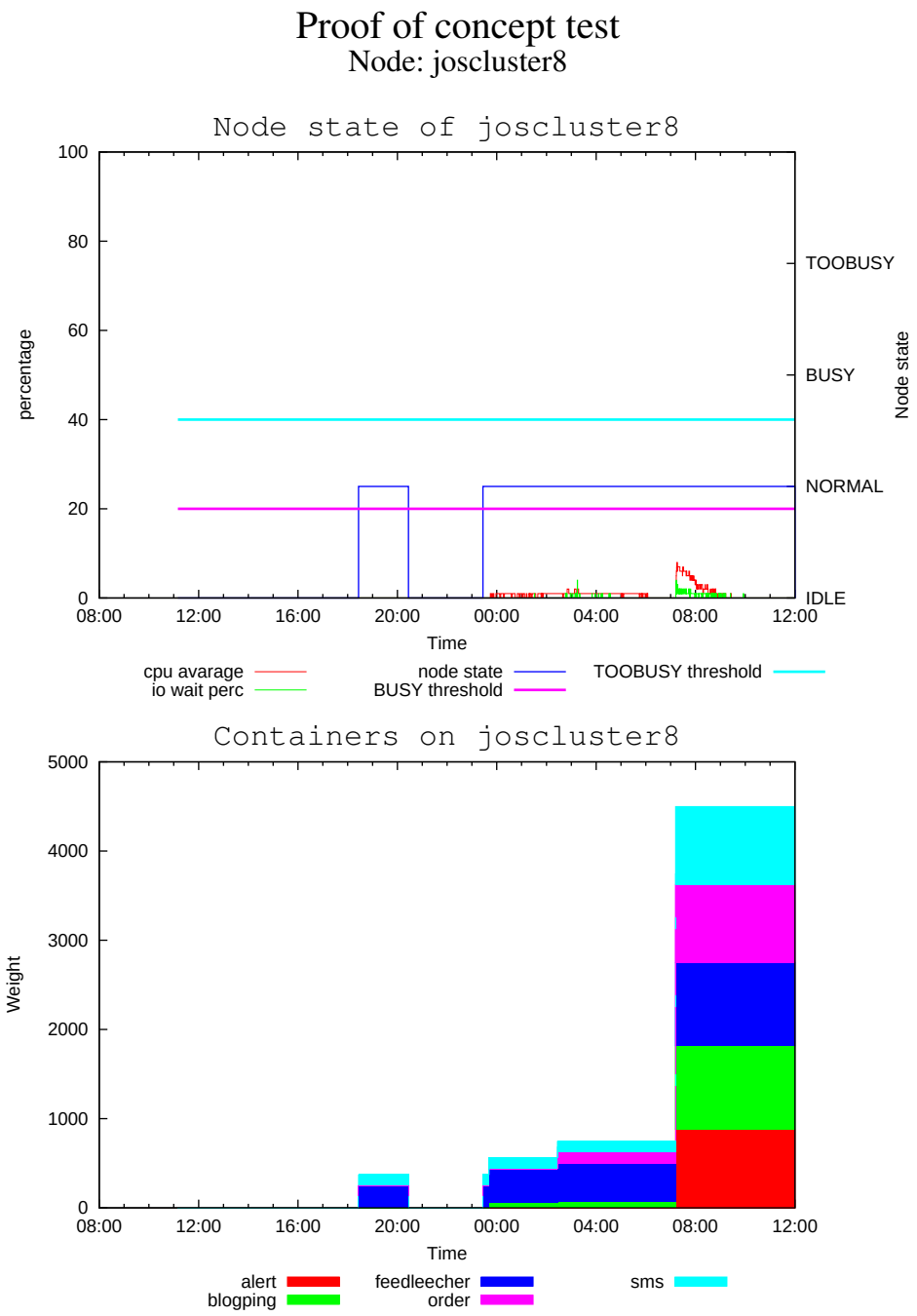

Figure B.12: Together with B.9, B.10, B.11 and B.13 this plot shows the state of the queue system. It shows cpu and node state in conjunction with the containers and there accumulated weight. The events inserted according to B.9 are distributed over the containers (B.10) and the containers in turn are distributed over the available nodes (B.11, B.12, B.13).

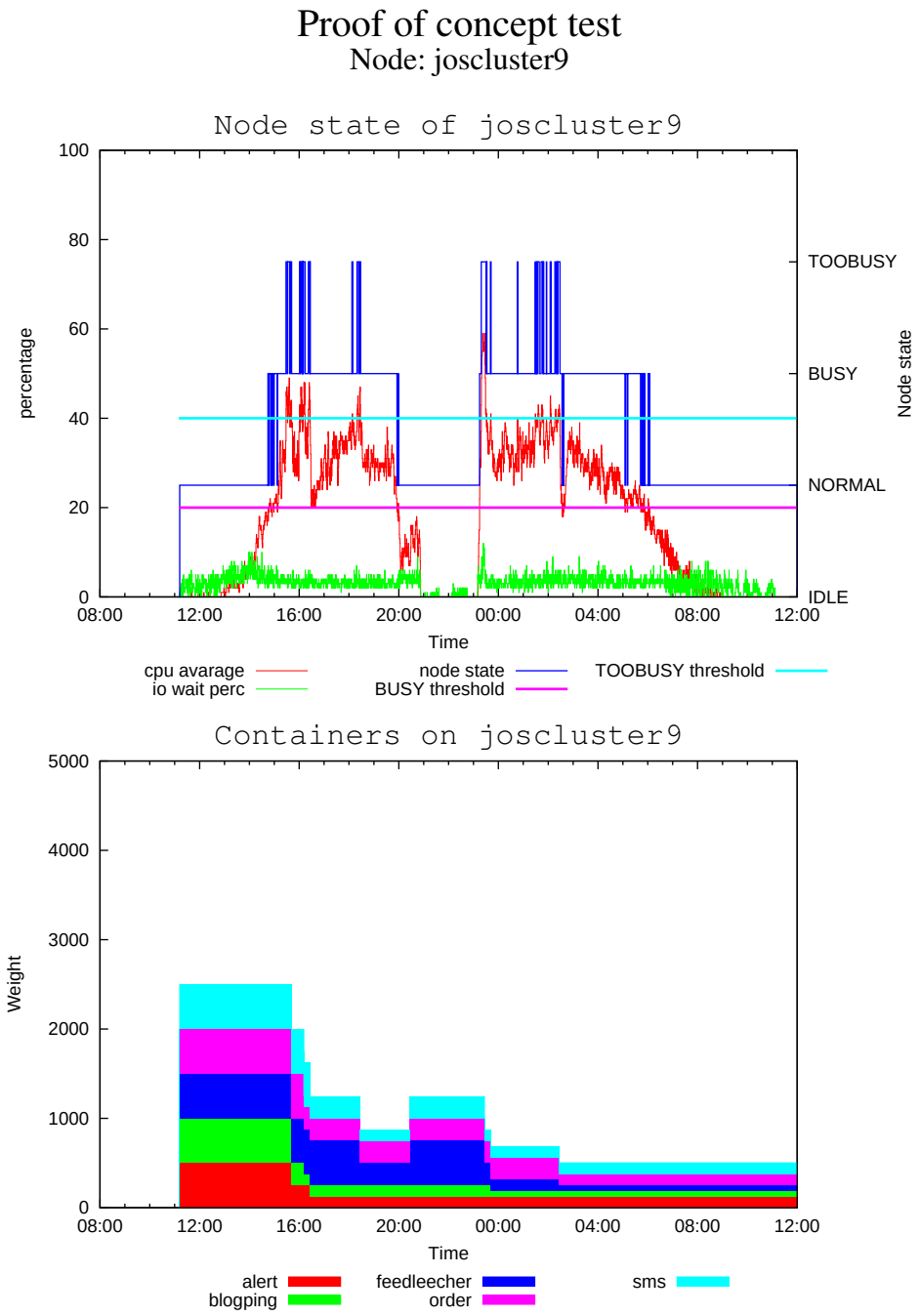

Figure B.13: Together with B.9, B.10, B.11 and B.12 this plot shows the state of the queue system. It shows cpu and node state in conjunction with the containers and there accumulated weight. The events inserted according to B.9 are distributed over the containers (B.10) and the containers in turn are distributed over the available nodes (B.11, B.12, B.13).

B.1.3 Standalone tests

B.1.3.1 Model - select only one container

This section contains the extra plots for the queue model test run that moved only one container each time, discussed in section 4.1.2 on page 36.

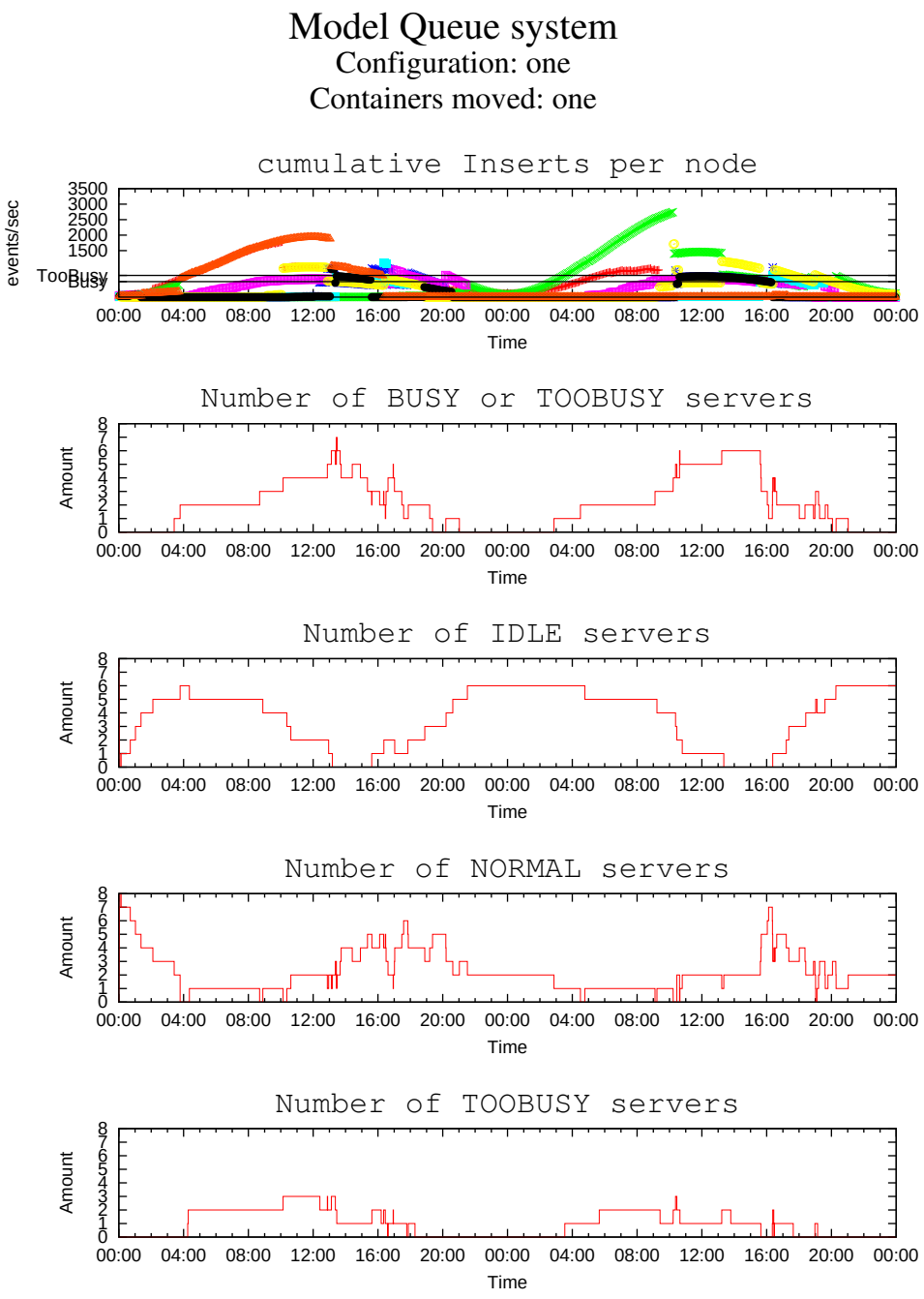

Figure B.14: Together with B.15, B.16 and B.17 this plot tests moving a single container in all four configurations. This series can be compared to the plots in B.1.1 which test moving 25% and 40% of the containers.

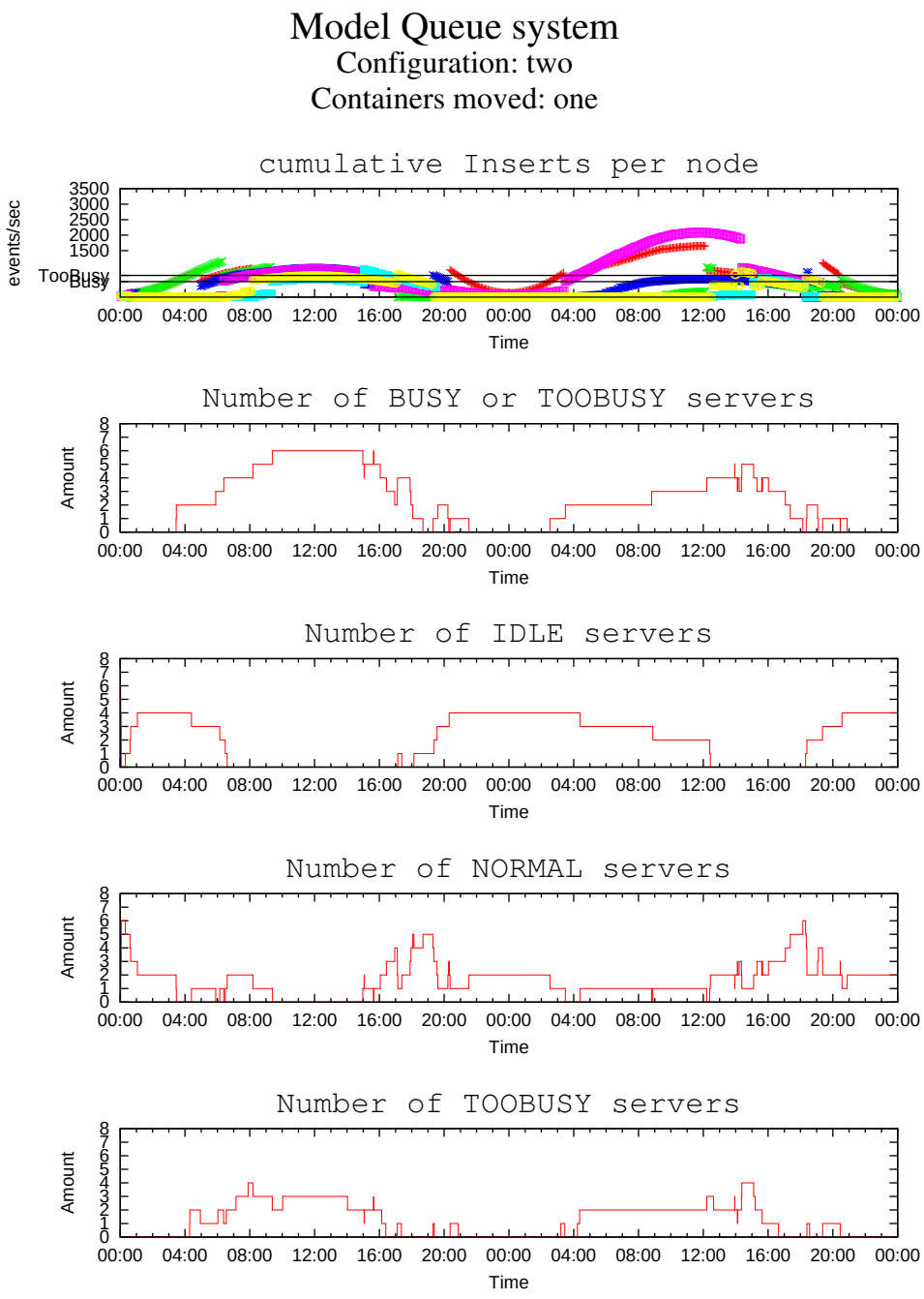

Figure B.15: Together with B.14, B.16 and B.17 this plot tests moving a single container in all four configurations. This series can be compared to the plots in B.1.1 which test moving 25% and 40% of the containers.

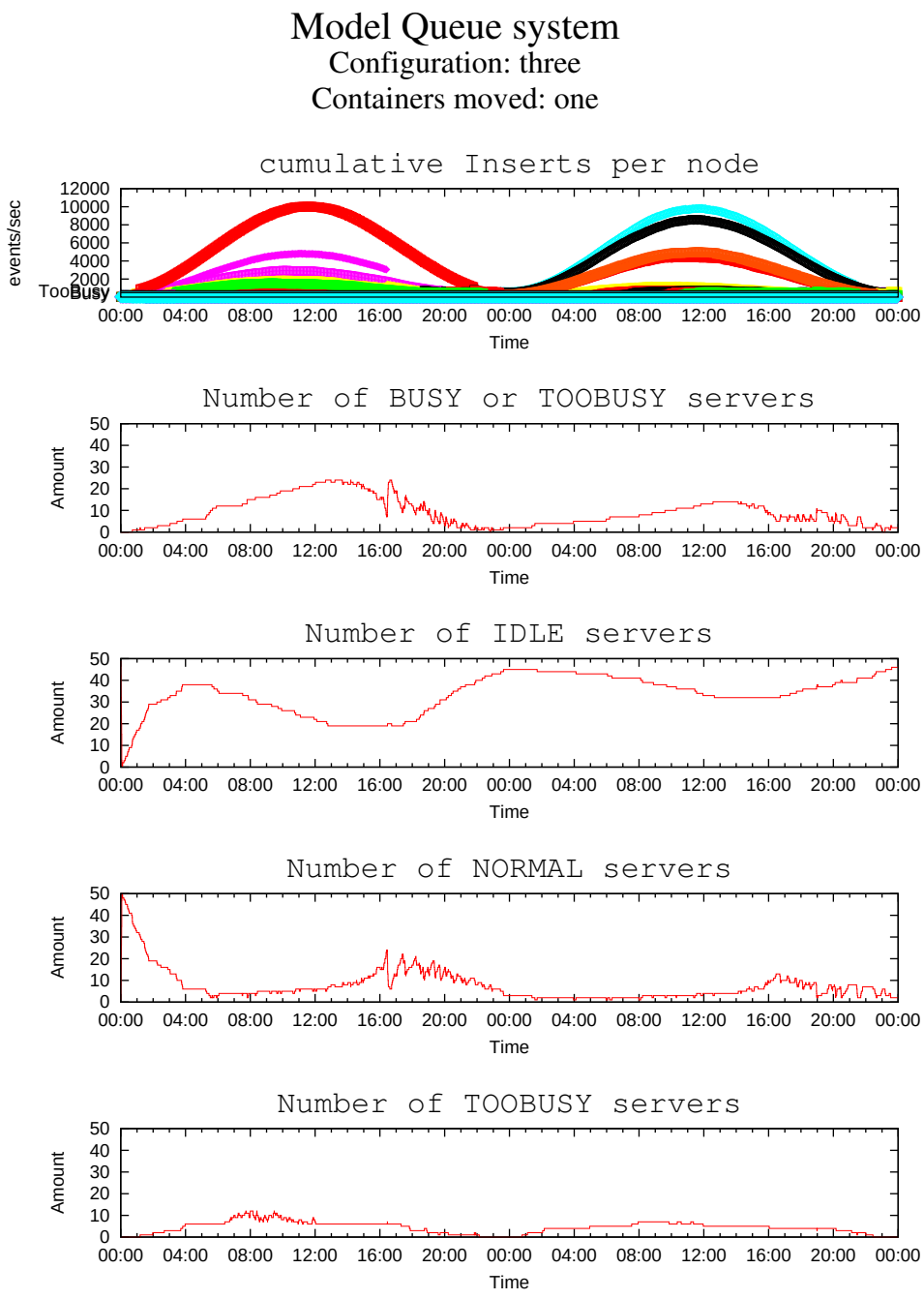

Figure B.16: Together with B.14, B.15 and B.17 this plot tests moving a single container in all four configurations. This series can be compared to the plots in B.1.1 which test moving 25% and 40% of the containers.

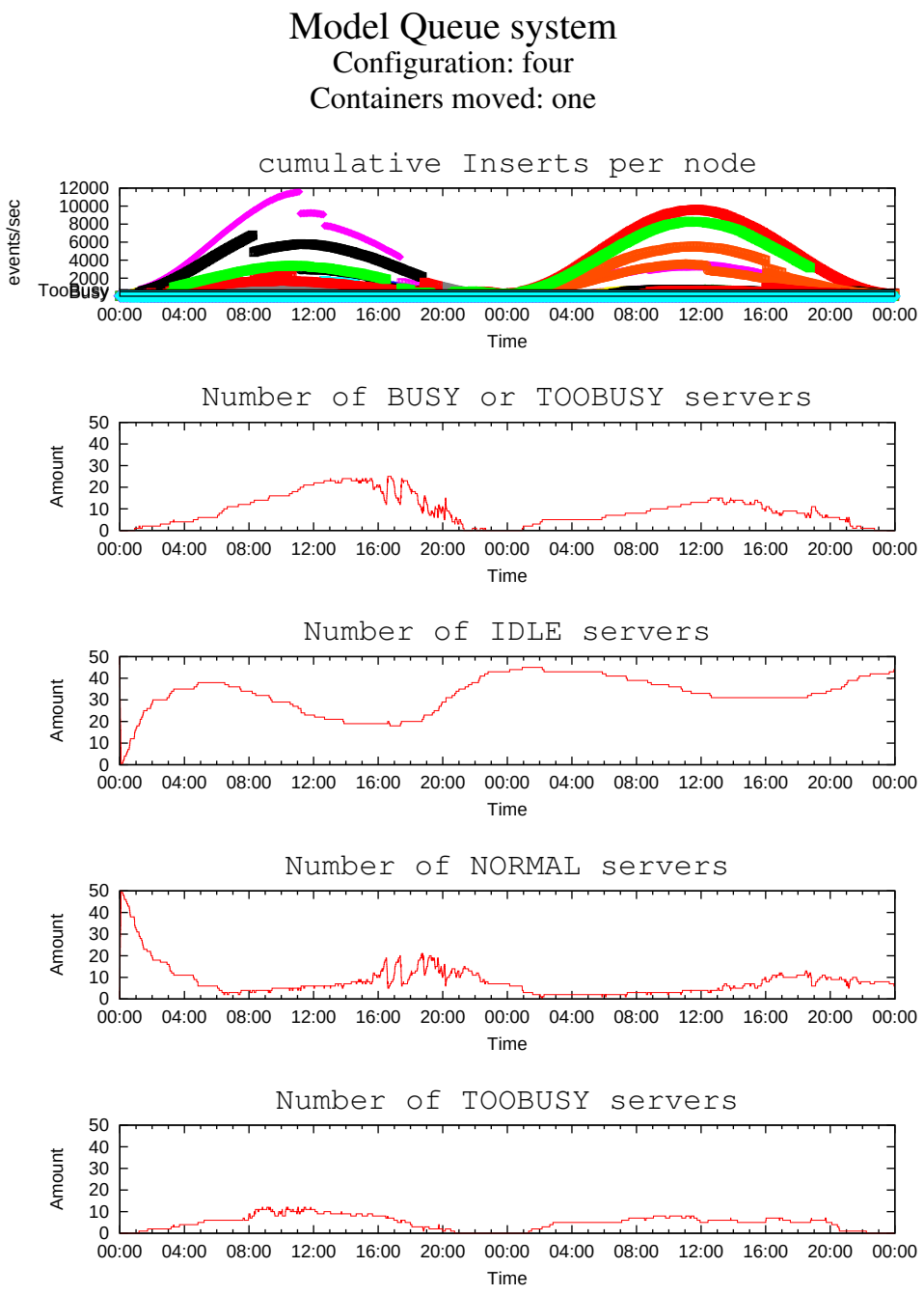

Figure B.17: Together with B.14, B.15 and B.16 this plot tests moving a single container in all four configurations. This series can be compared to the plots in B.1.1 which test moving 25% and 40% of the containers.

B.1.3.2 Crash

This section contains the plots for the queue system crash test, discussed in section 4.1.3.3 on page 39.

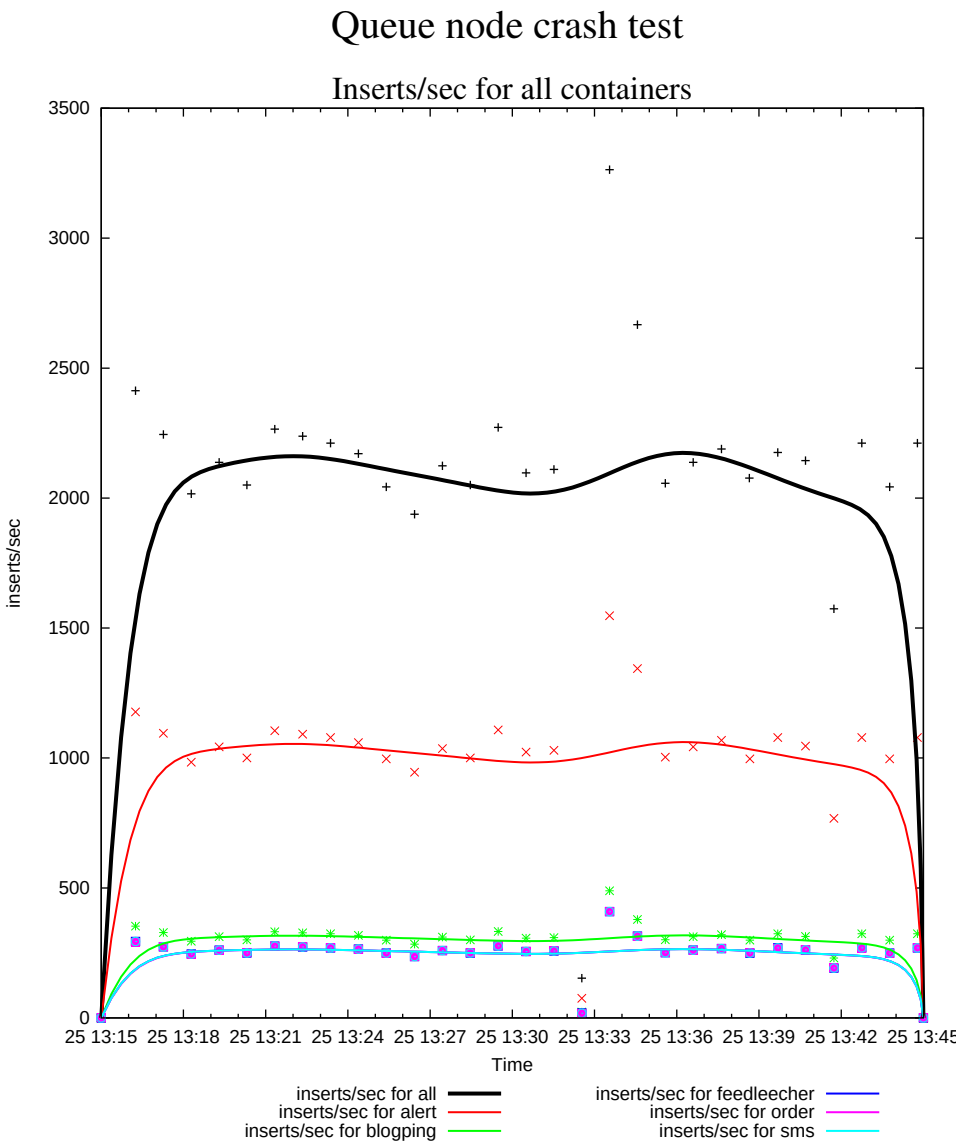

Figure B.18: Together with B.19, B.20 and B.21 this plot shows the reaction of the system on the crash of joscluster8 around 13:35

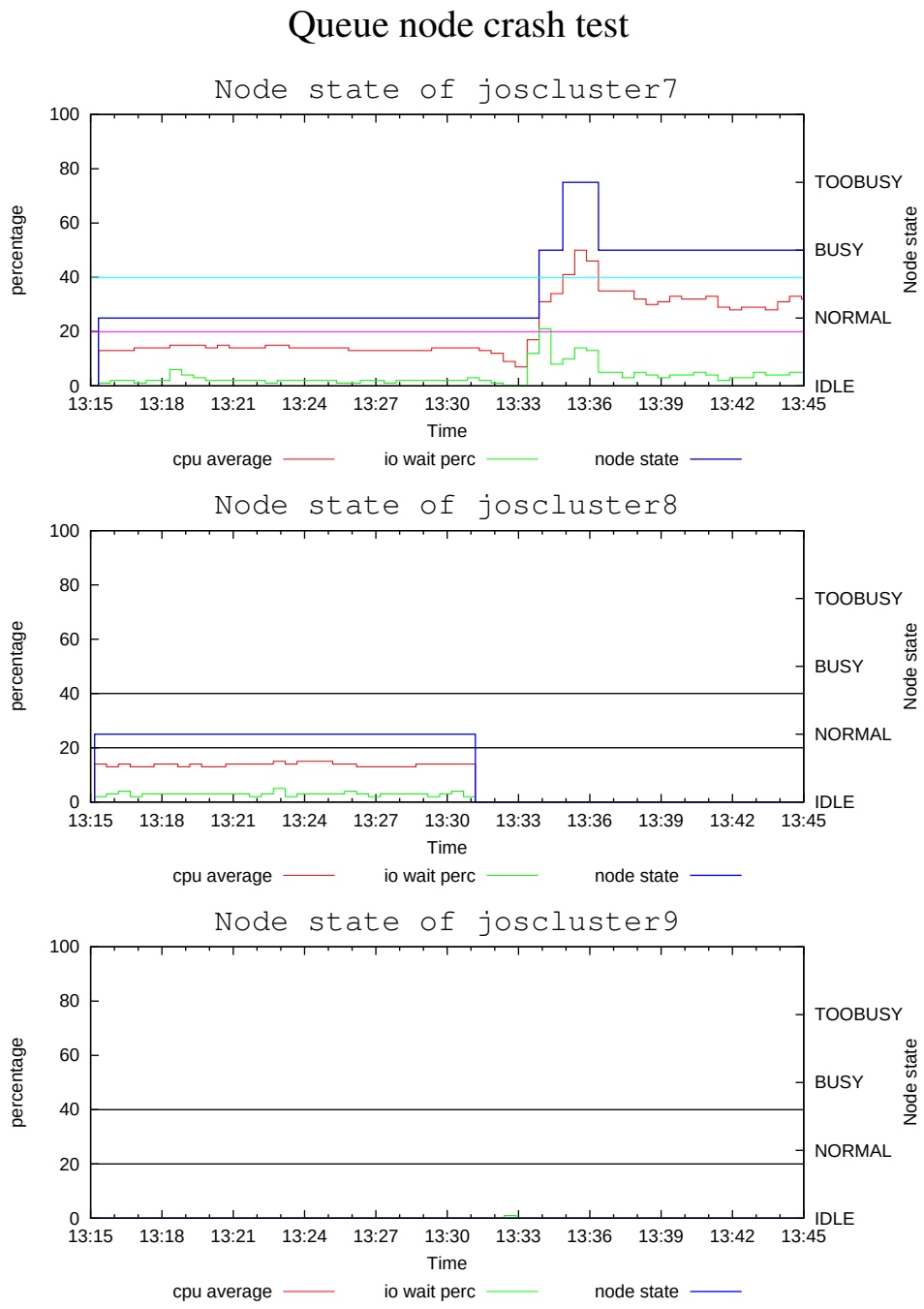

Figure B.19: Together with B.18, B.20 and B.21 this plot shows the reaction of the system on the crash of joscluster8 around 13:35

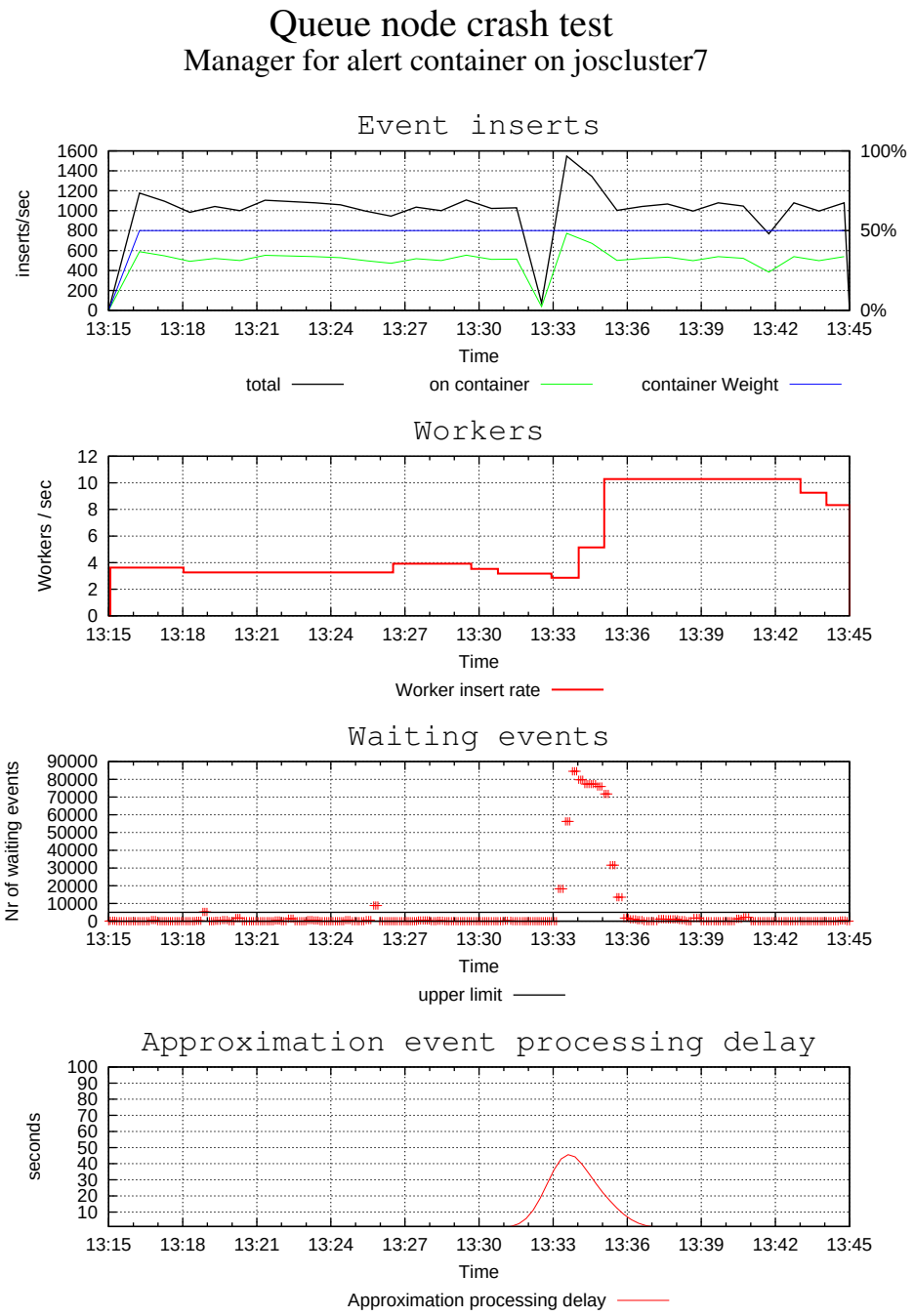

Figure B.20: Together with B.18, B.19 and B.21 this plot shows the reaction of the system on the crash of joscluster8 around 13:35. It shows that the manager has to raise its worker insert rate in response to the increased load

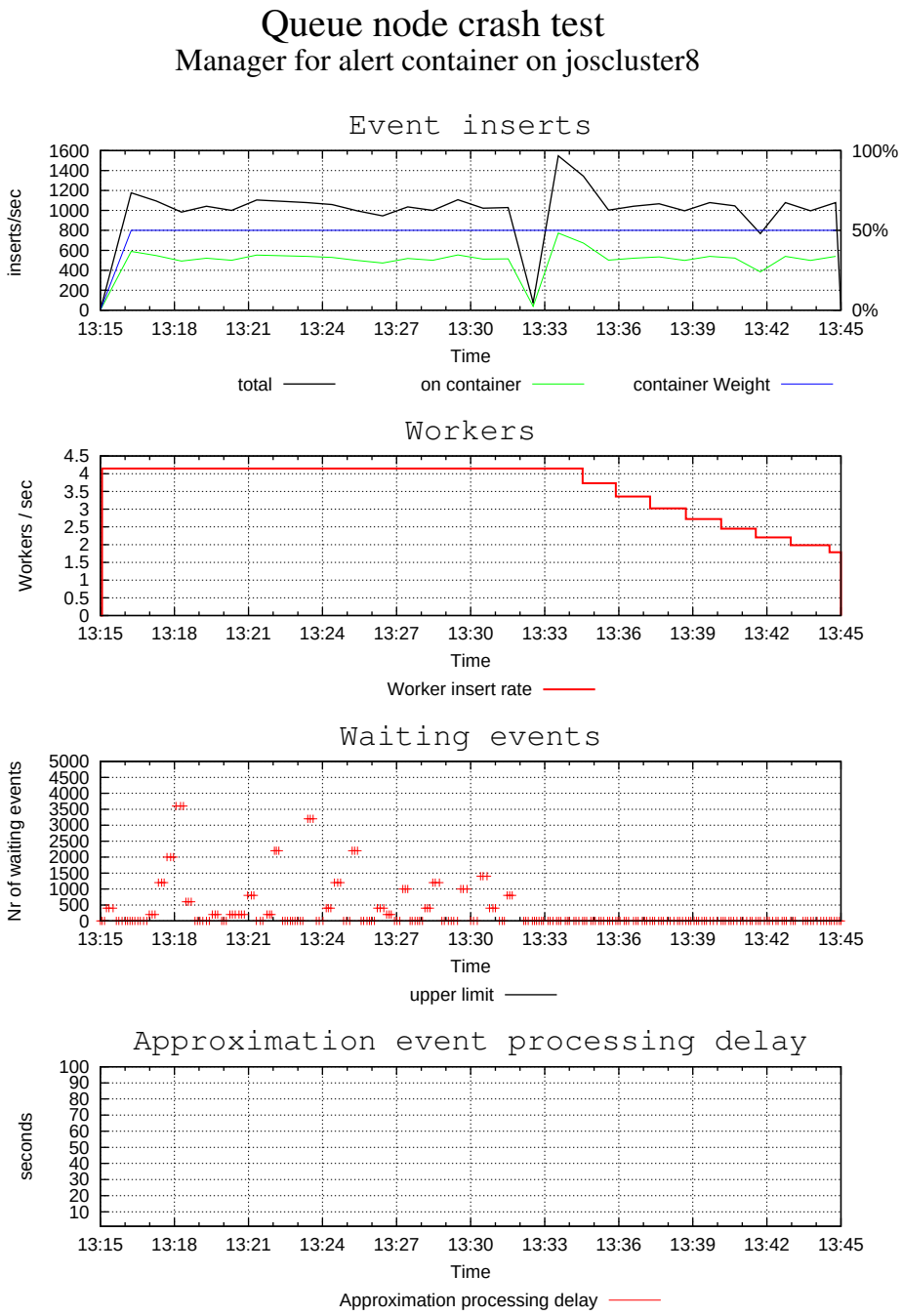

Figure B.21: Together with B.18, B.19 and B.21 this plot shows the reaction of the system on the crash of joscluster8 around 13:35. It shows that the alert manager for joscluster8 does not crash at the same time as joscluster8. It however does lower its worker insert rate because it can no longer connect to its queue. Eventually the process is killed because its submit node (joscluster8) went down.

B.2 Consumer/worker system

This section contains three kind of plots. Plots from the model test run, the PoC test run and special crafted tests/plots to tests different aspects.

B.2.1 Model test run

This section contains all plots from the model test run. They are ordered by insert rate and then by the applied upper limit. For a discussion of the results see section 4.2.2.3 on page 44.

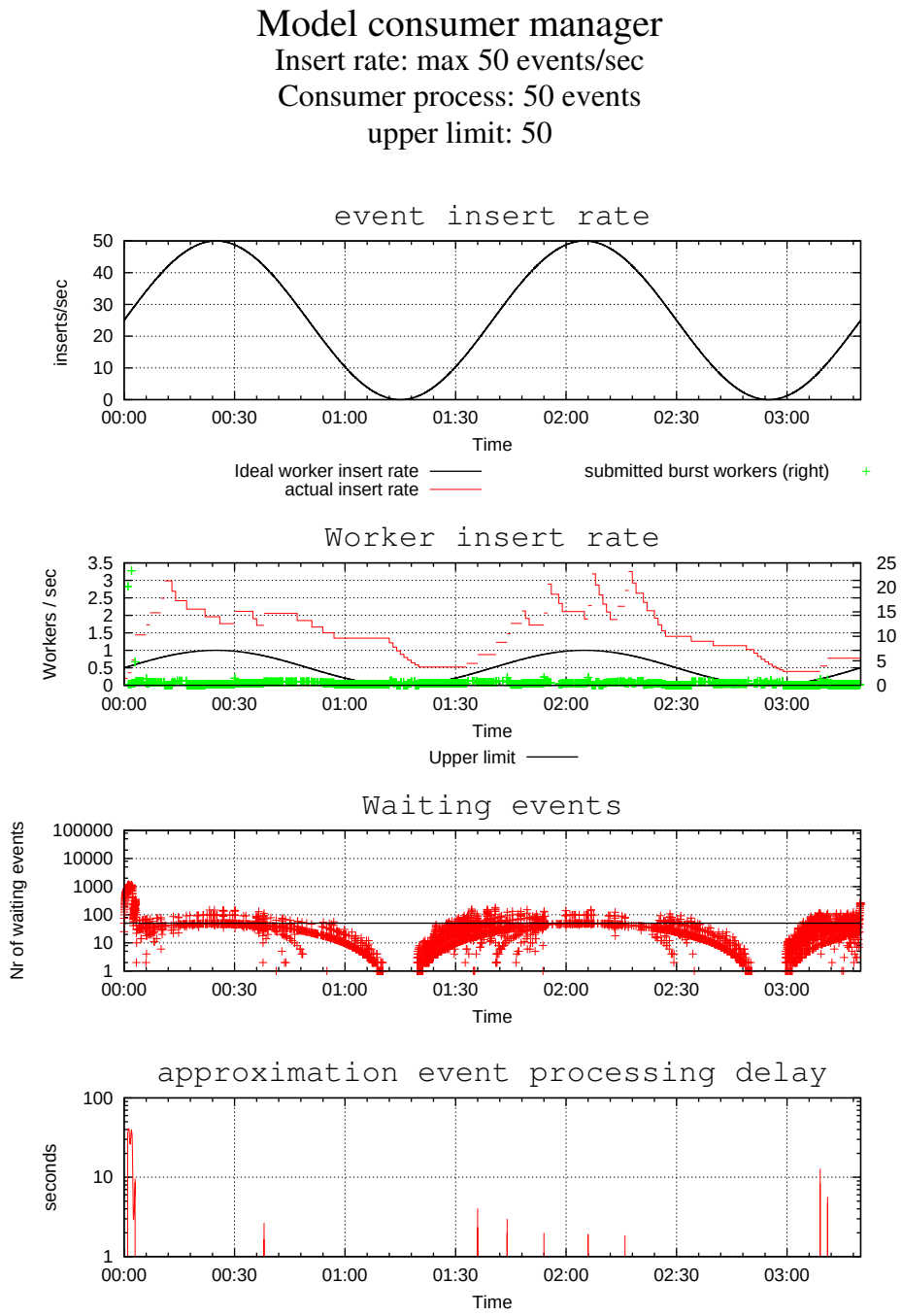

Figure B.22: The plot for the consumer worker model with a varying insert rate of max 50 events per second and an upper limit of 50. The consumer process 50 events per run. Together with B.23, B.24, B.25, B.26, B.27 and B.28 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average.

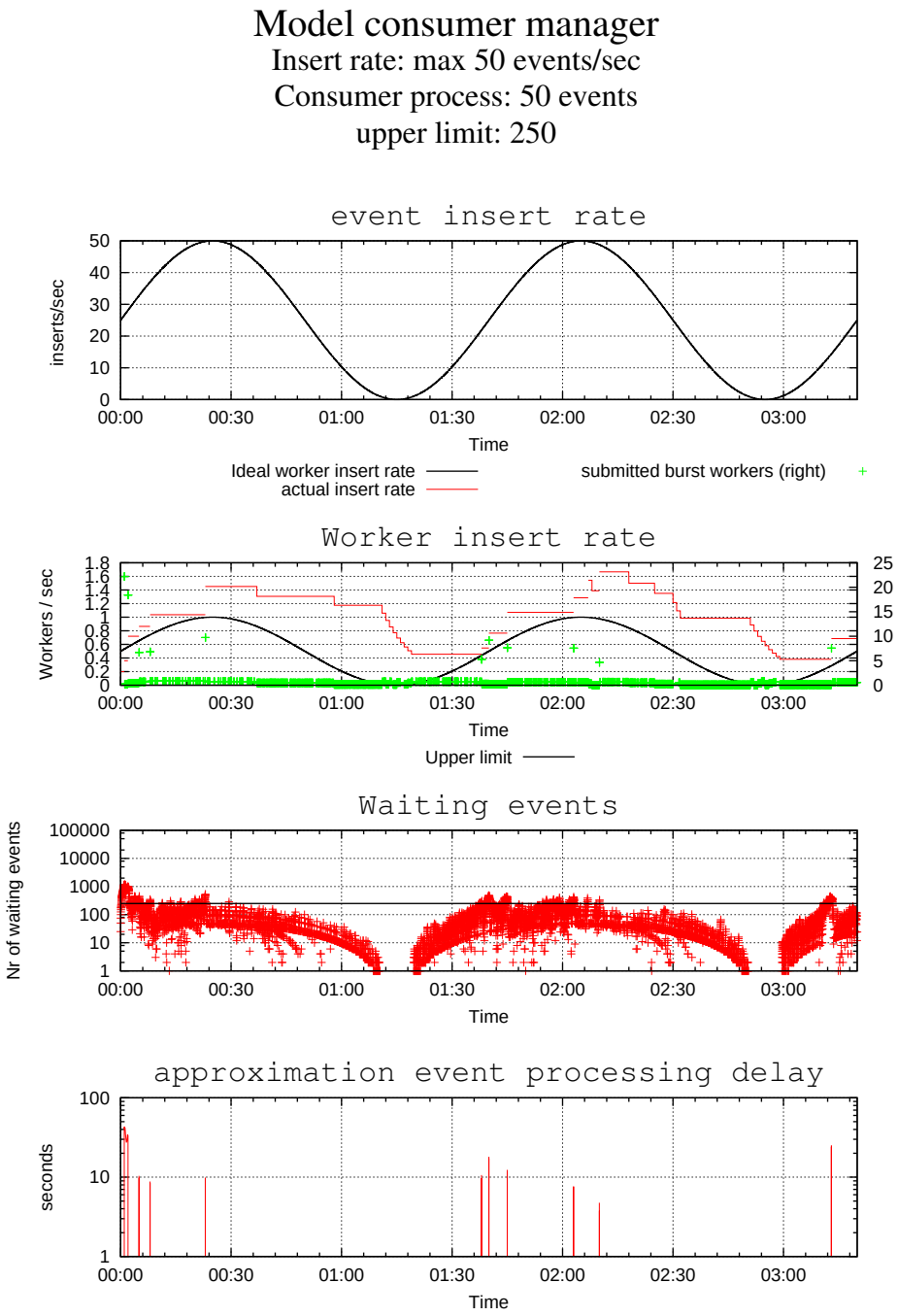

Figure B.23: As B.22, upper limit adjusted to 250. Together with B.22, B.24, B.25, B.26, B.27 and B.28 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average.

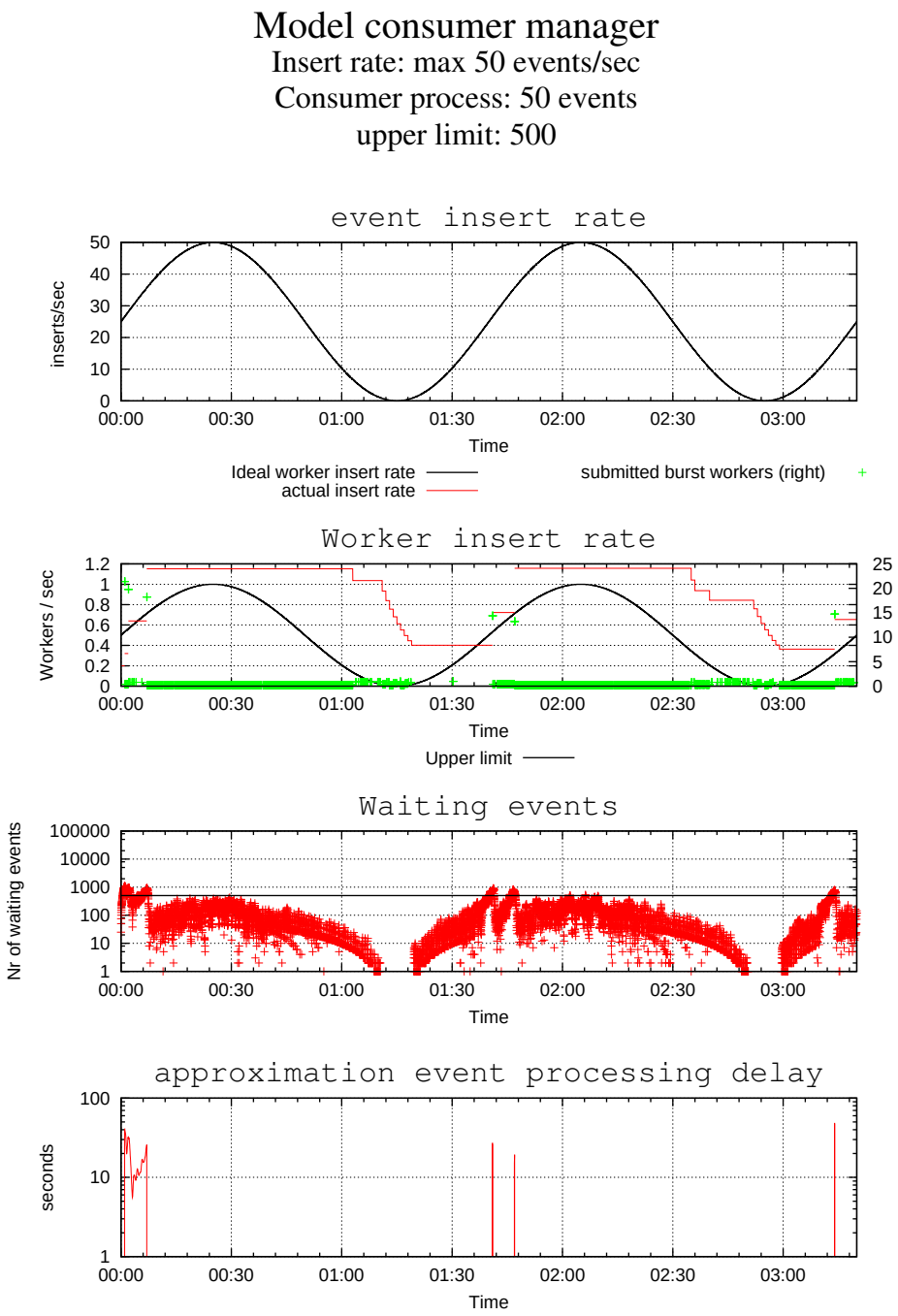

Figure B.24: As B.22, upper limit adjusted to 500. Together with B.22, B.23, B.25, B.26, B.27 and B.28 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average.

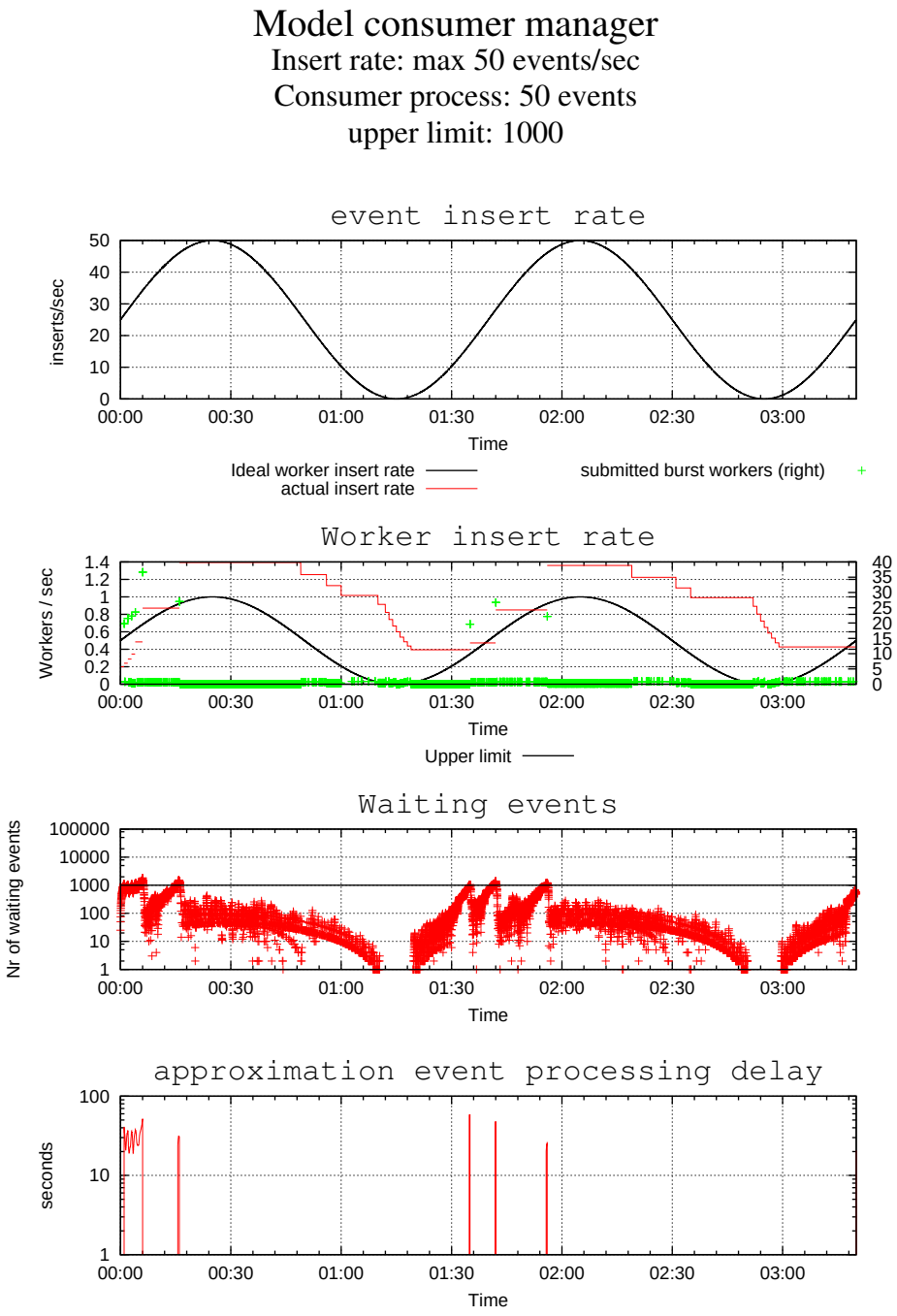

Figure B.25: As B.22, upper limit adjusted to 1000. Together with B.22, B.23, B.24, B.26, B.27 and B.28 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average.

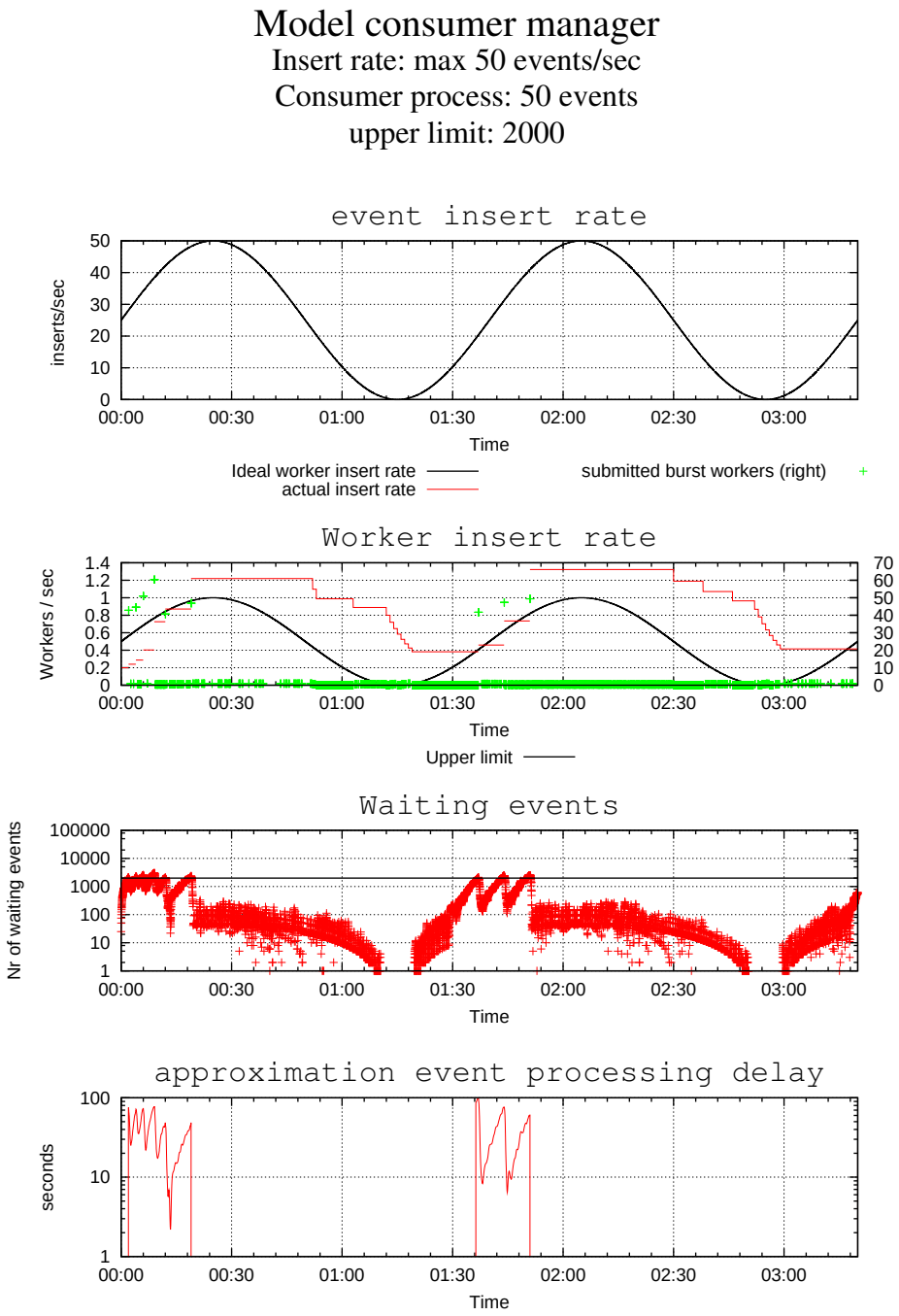

Figure B.26: As B.22, upper limit adjusted to 2000. Together with B.22, B.23, B.24, B.25, B.27 and B.28 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average.

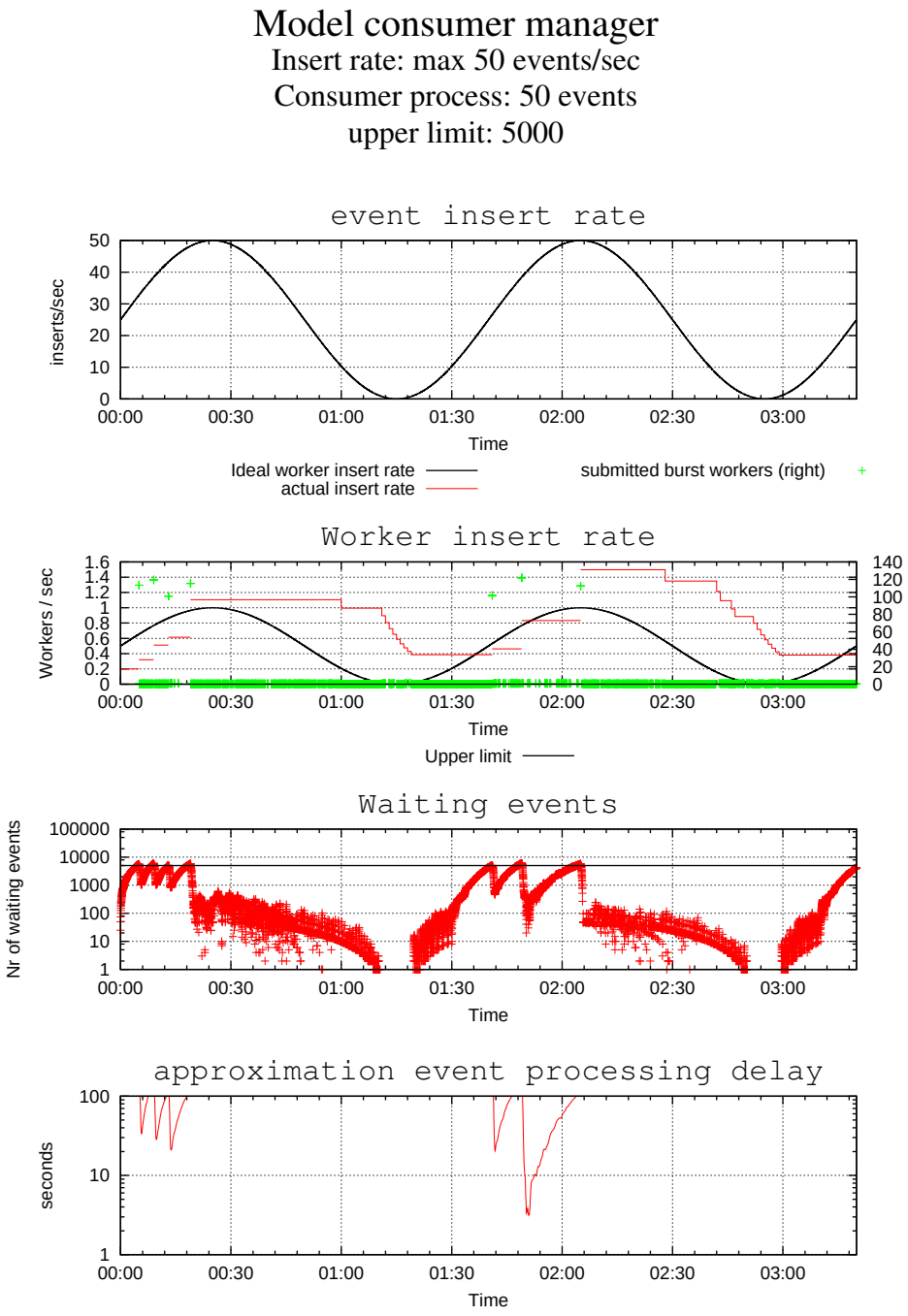

Figure B.27: As B.22, upper limit adjusted to 5000. Together with B.22, B.23, B.24, B.25, B.26 and B.28 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average.

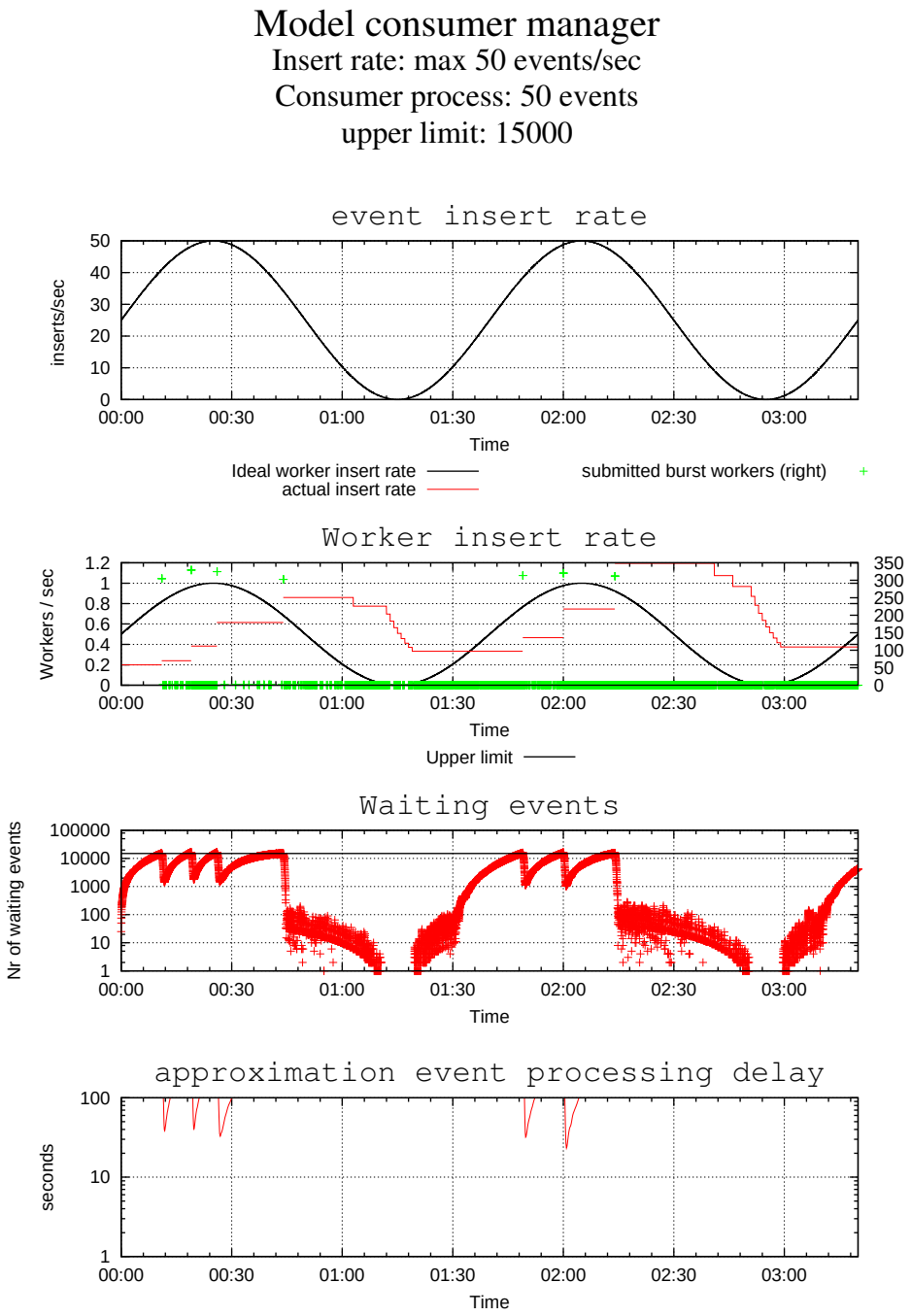

Figure B.28: As B.22, upper limit adjusted to 250. Together with B.22, B.23, B.24, B.25, B.26 and B.27 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average.

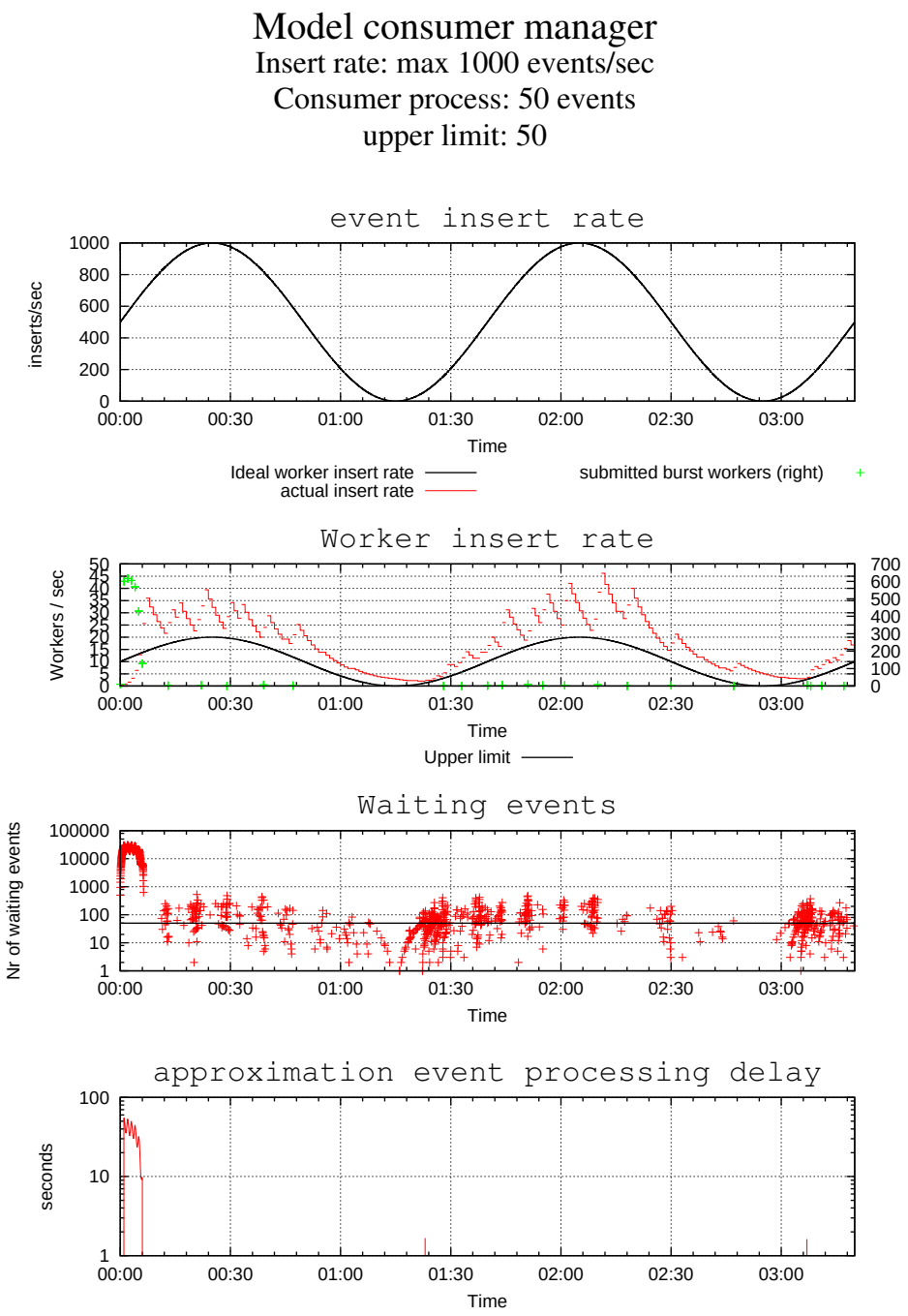

Figure B.29: The plot for the consumer worker model with a varying insert rate of max 1000 events per second and an upper limit of 50. The consumer process 50 events per run. Together with B.30, B.31, B.32, B.33, B.34 and B.35 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently.

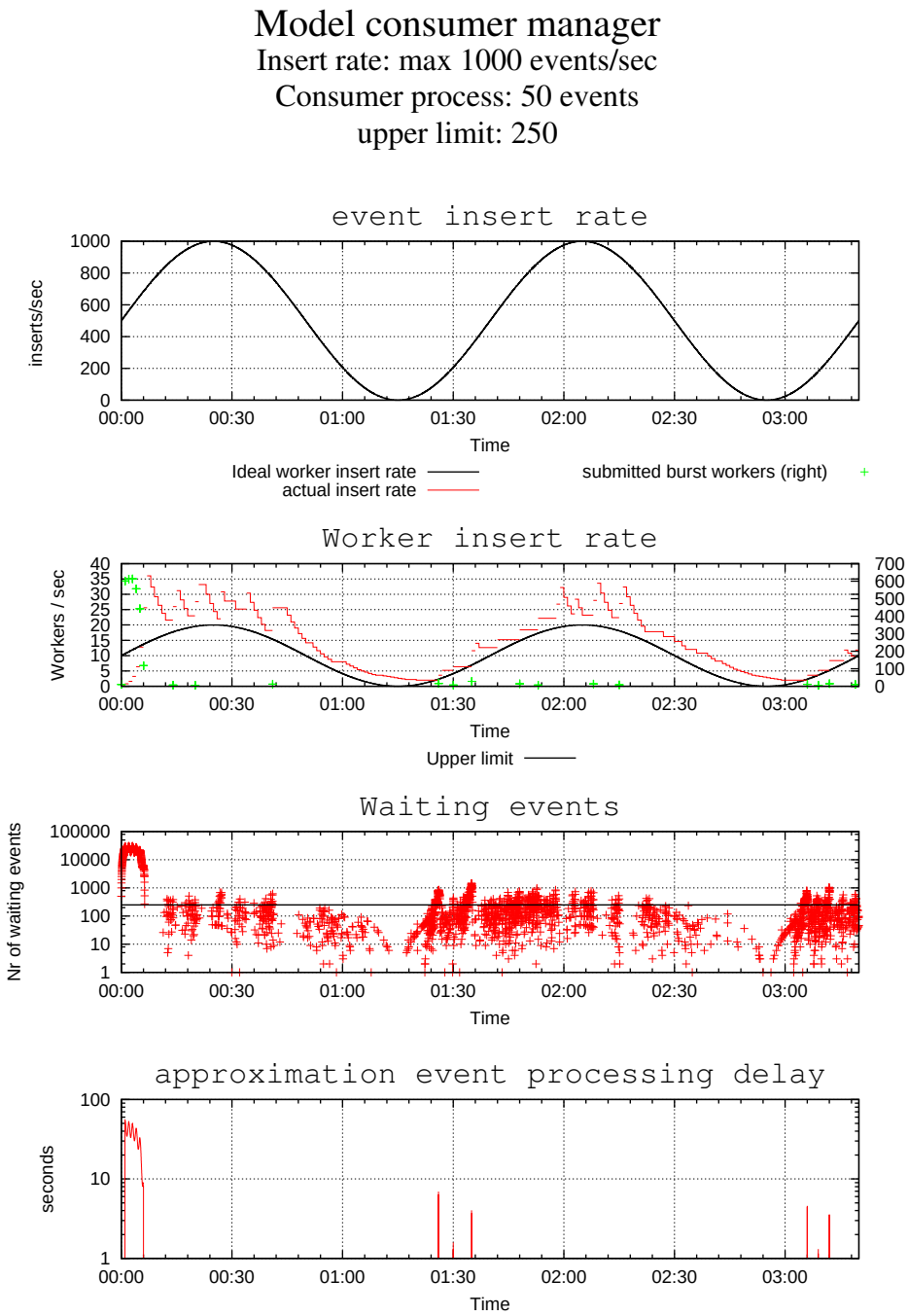

Figure B.30: As B.29, upper limit adjusted to 250. Together with B.30, B.31, B.32, B.33, B.34 and B.35 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently.

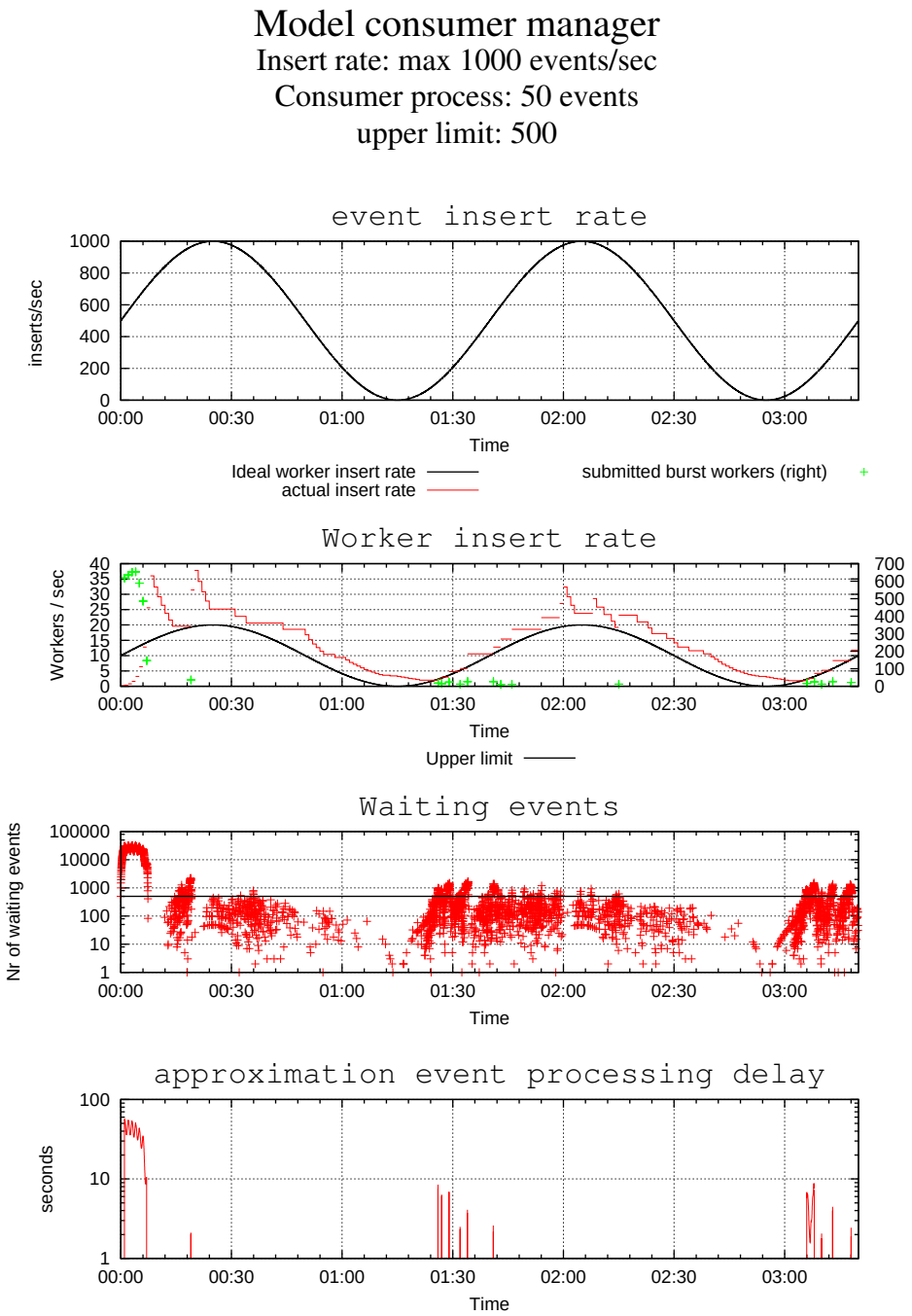

Figure B.31: As B.29, upper limit adjusted to 500. Together with B.30, B.31, B.32, B.33, B.34 and B.35 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently.

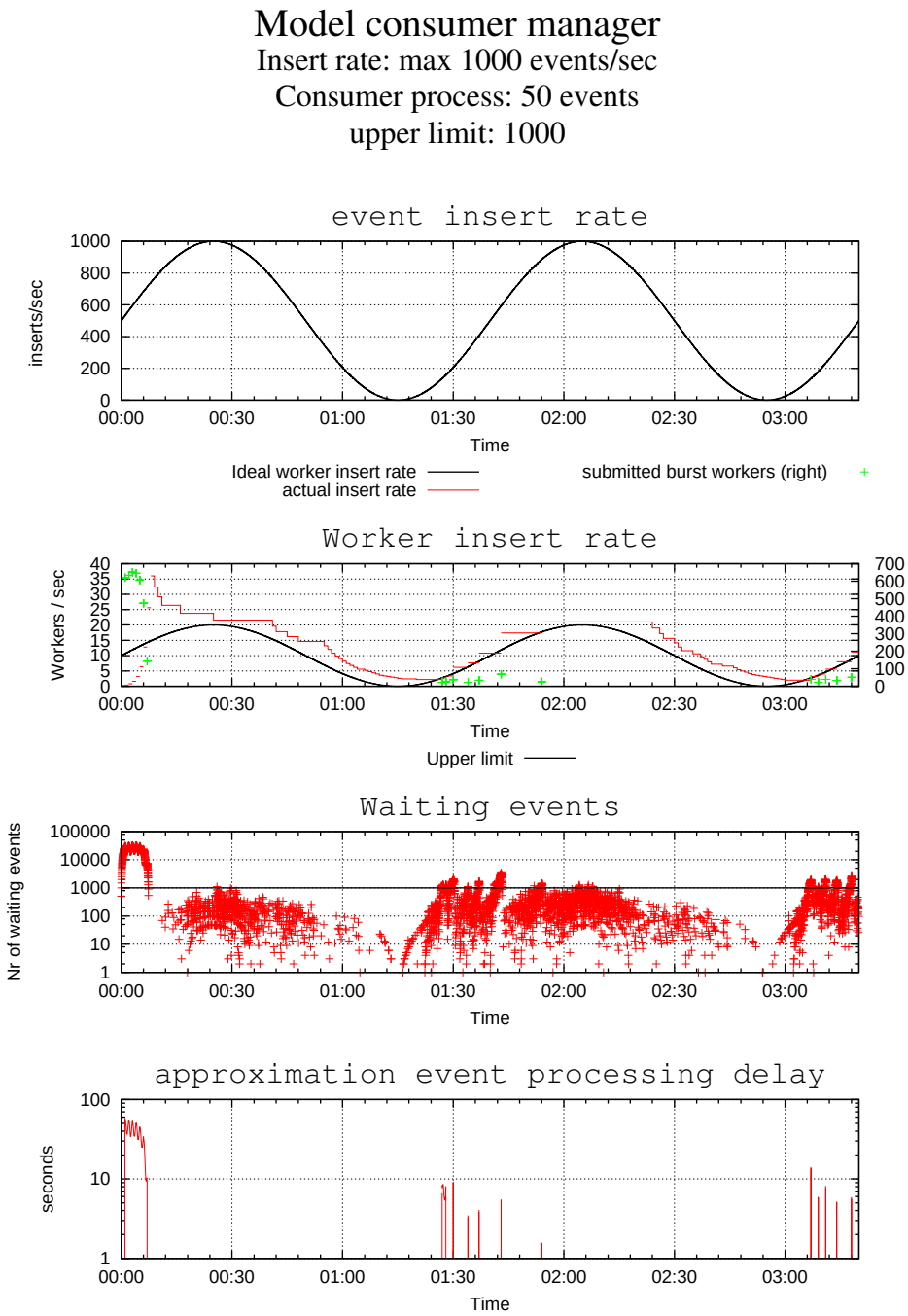

Figure B.32: As B.29, upper limit adjusted to 1000. Together with B.30, B.31, B.32, B.33, B.34 and B.35 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently.

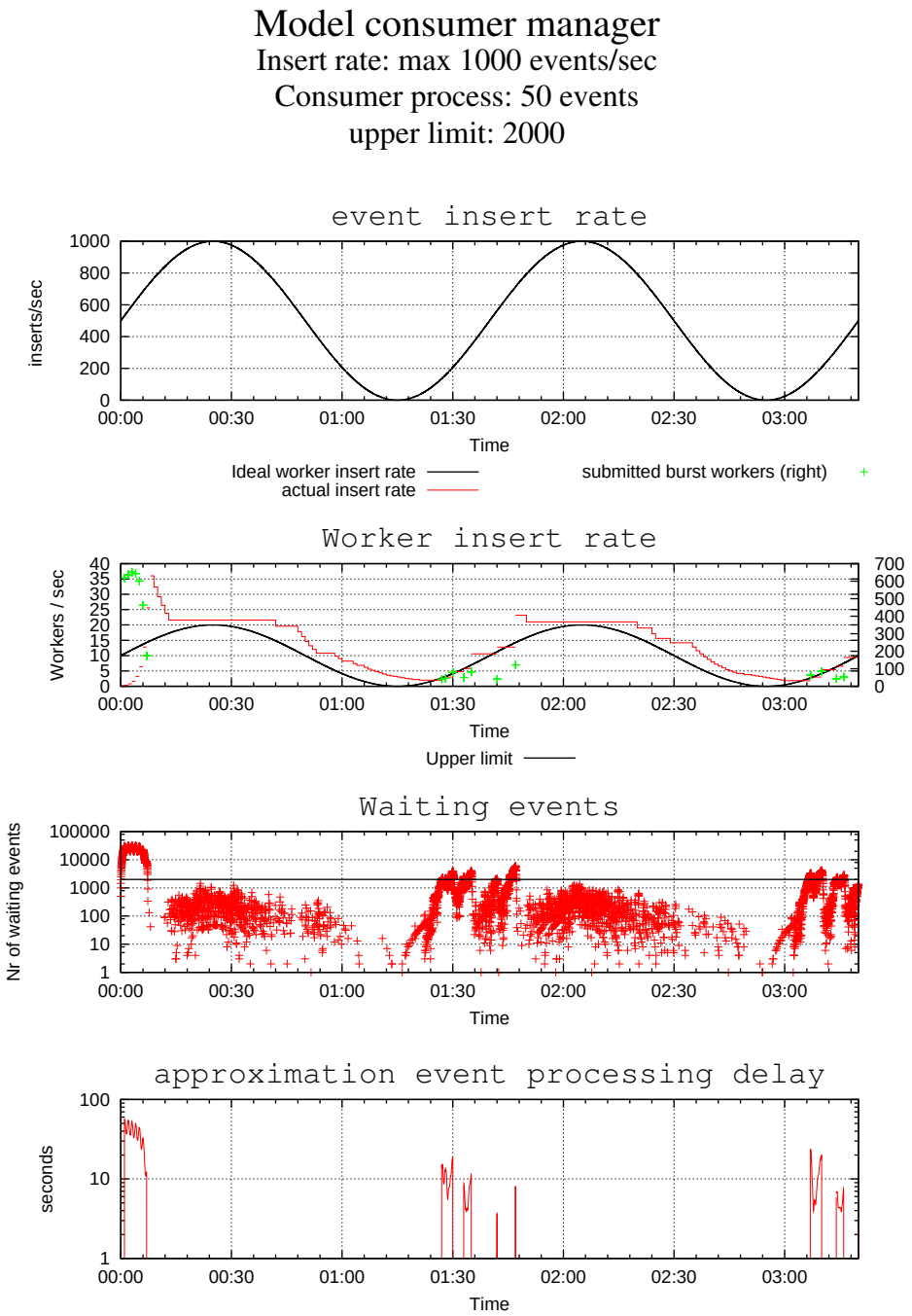

Figure B.33: As B.29, upper limit adjusted to 2000. Together with B.30, B.31, B.32, B.33, B.34 and B.35 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently.

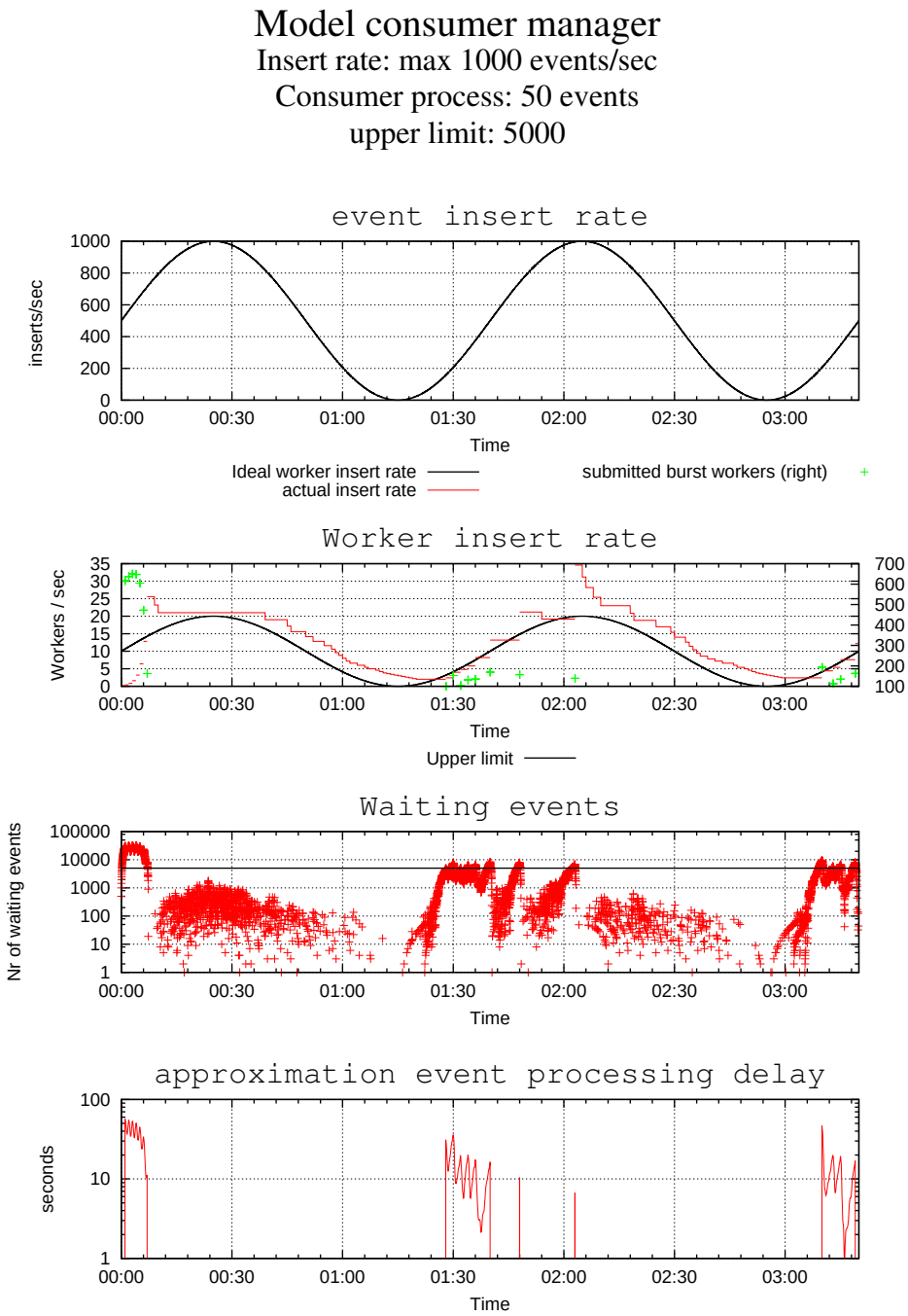

Figure B.34: As B.29, upper limit adjusted to 5000. Together with B.30, B.31, B.32, B.33, B.34 and B.35 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently.

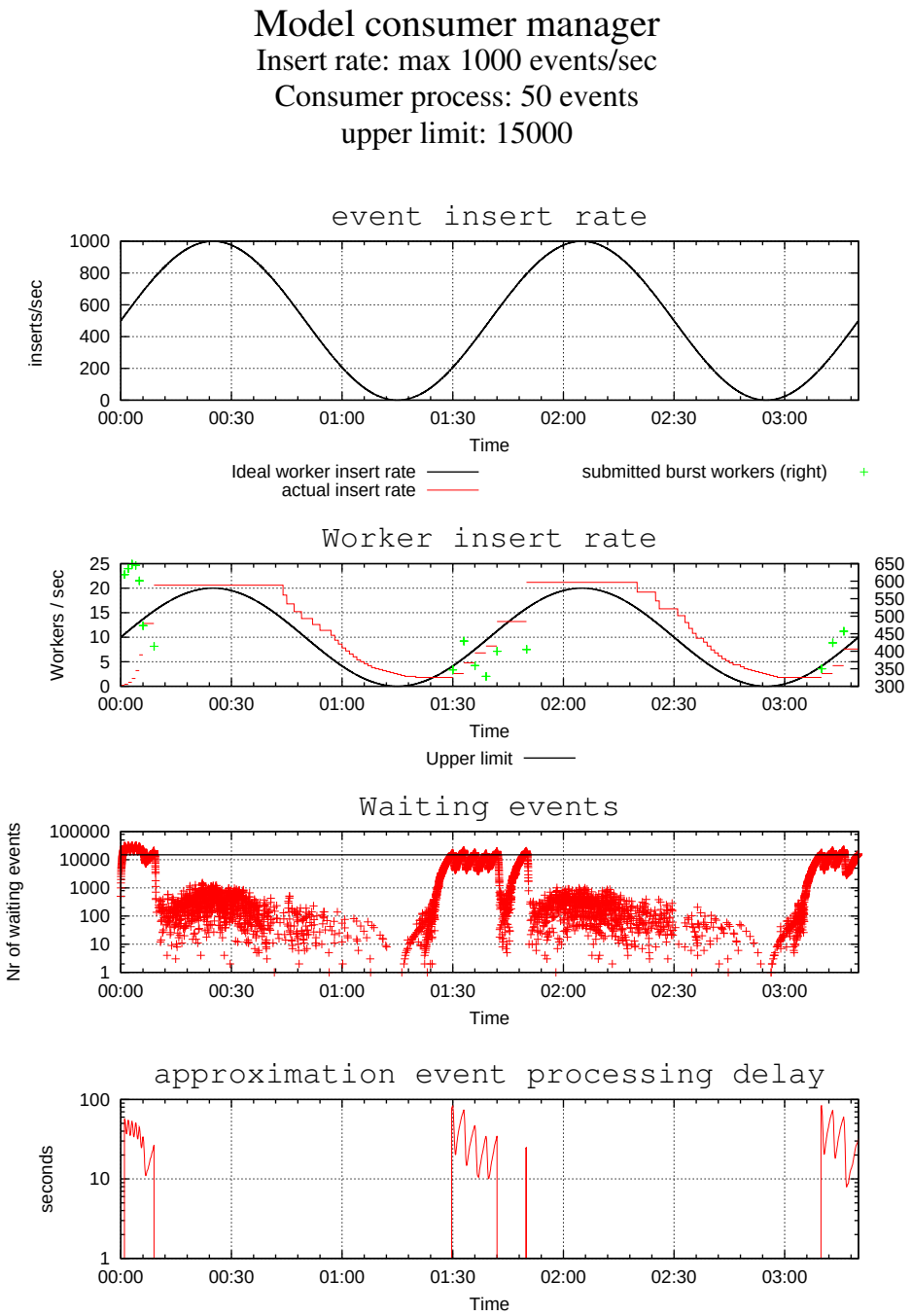

Figure B.35: As B.29, upper limit adjusted to 15000. Together with B.30, B.31, B.32, B.33, B.34 and B.35 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently.

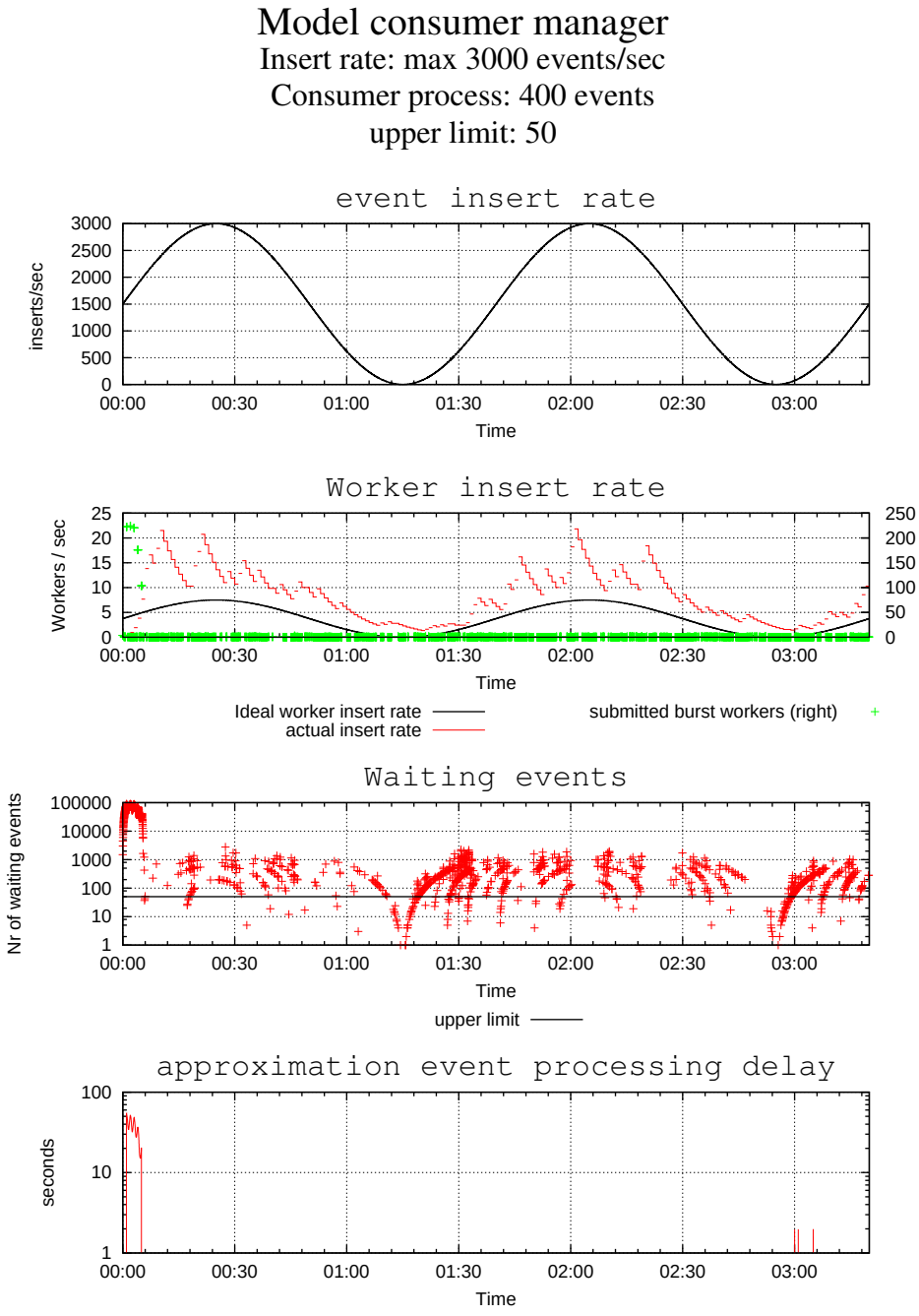

Figure B.36: The plot for the consumer worker model with a varying insert rate of max 3000 events per second and an upper limit of 50. The consumer process 400 events per run. These are the settings employed in the Proof of Concept solution. Together with B.37, B.38, B.39, B.40, B.41 and B.42 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently. A comparison with the series started by B.29 is hard to make, but waiting events seem to be more spread. A more in depth discussion is held in section 4.2.2.3 on page 44

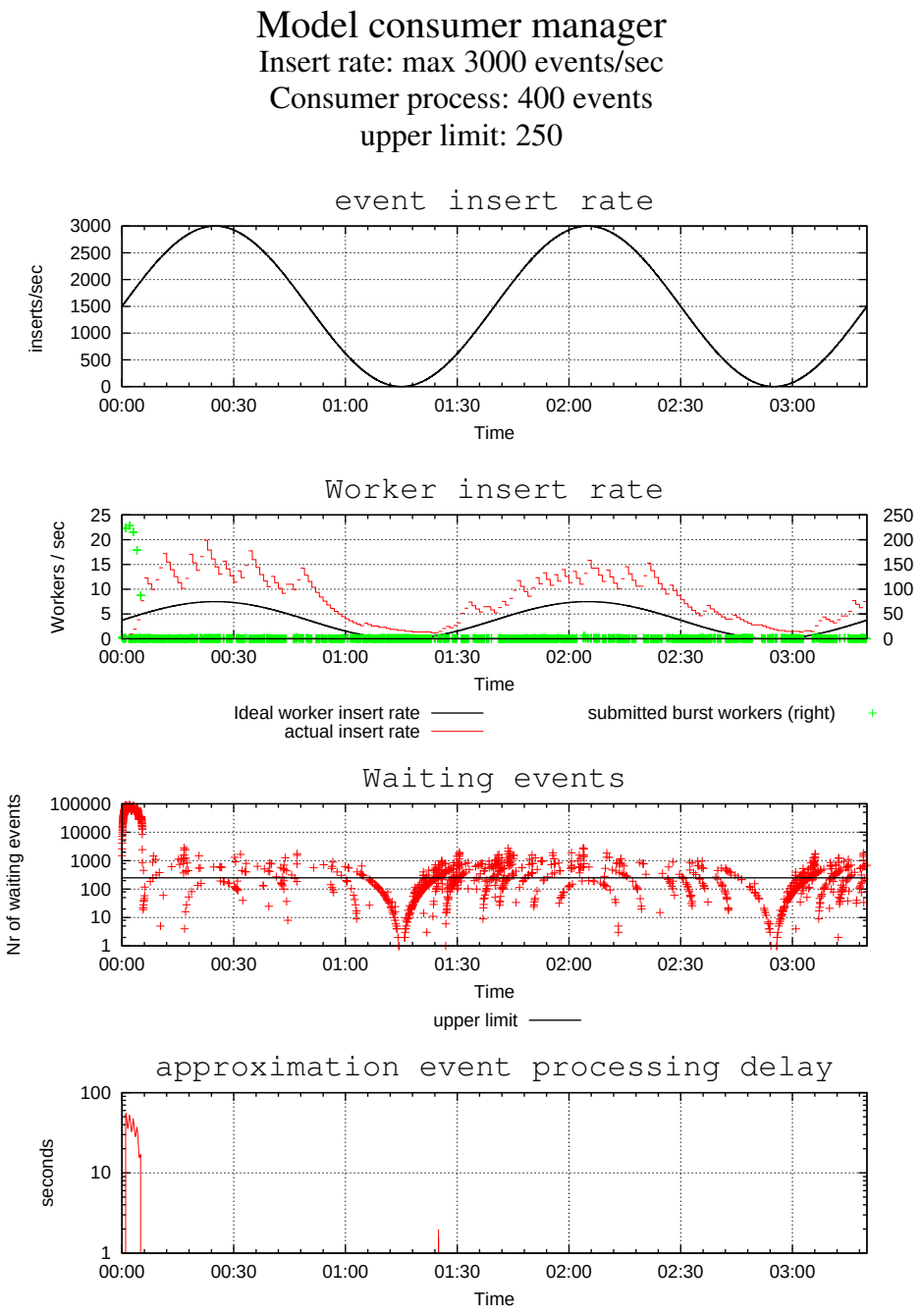

Figure B.37: As B.36, upper limit adjusted to 250. These are the settings employed in the Proof of Concept solution. Together with B.37, B.38, B.39, B.40, B.41 and B.42 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently. A comparison with the series started by B.29 is hard to make, but waiting events seem to be more spread. A more in depth discussion is held in section 4.2.2.3 on page 44

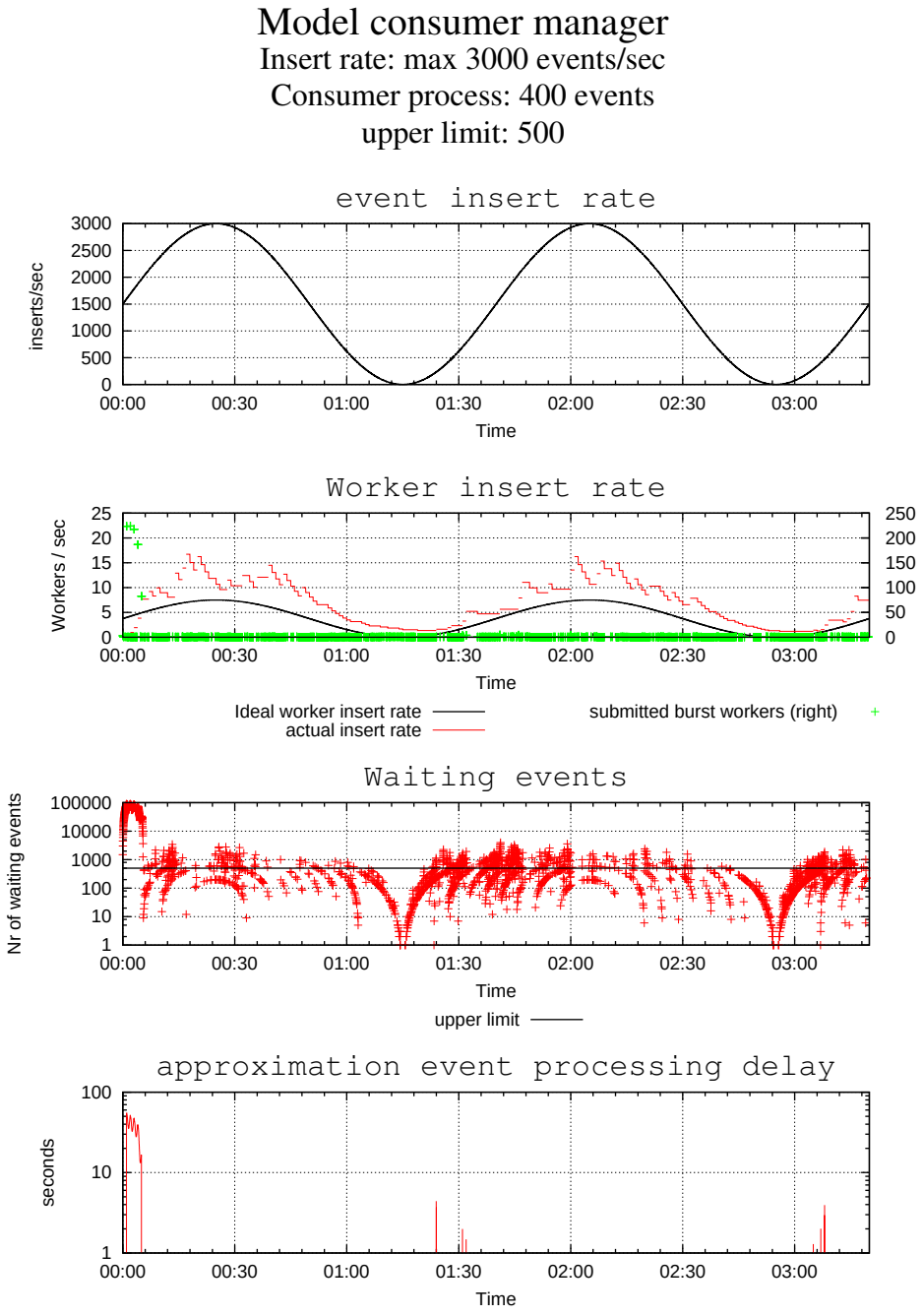

Figure B.38: As B.36, upper limit adjusted to 500. These are the settings employed in the Proof of Concept solution. Together with B.37, B.38, B.39, B.40, B.41 and B.42 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently. A comparison with the series started by B.29 is hard to make, but waiting events seem to be more spread. A more in depth discussion is held in section 4.2.2.3 on page 44

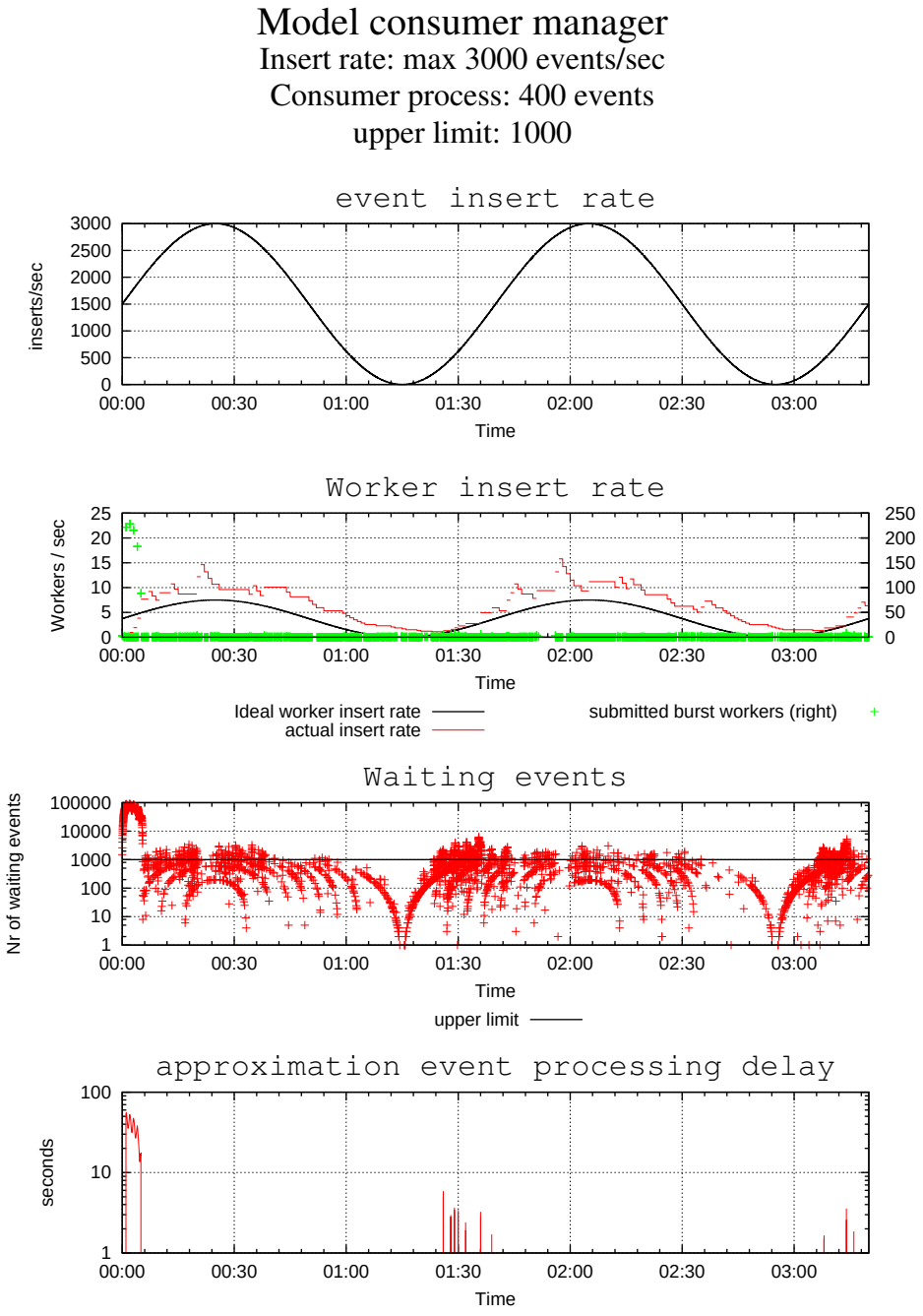

Figure B.39: As B.36, upper limit adjusted to 1000. These are the settings employed in the Proof of Concept solution. Together with B.37, B.38, B.39, B.40, B.41 and B.42 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently. A comparison with the series started by B.29 is hard to make, but waiting events seem to be more spread. A more in depth discussion is held in section 4.2.2.3 on page 44

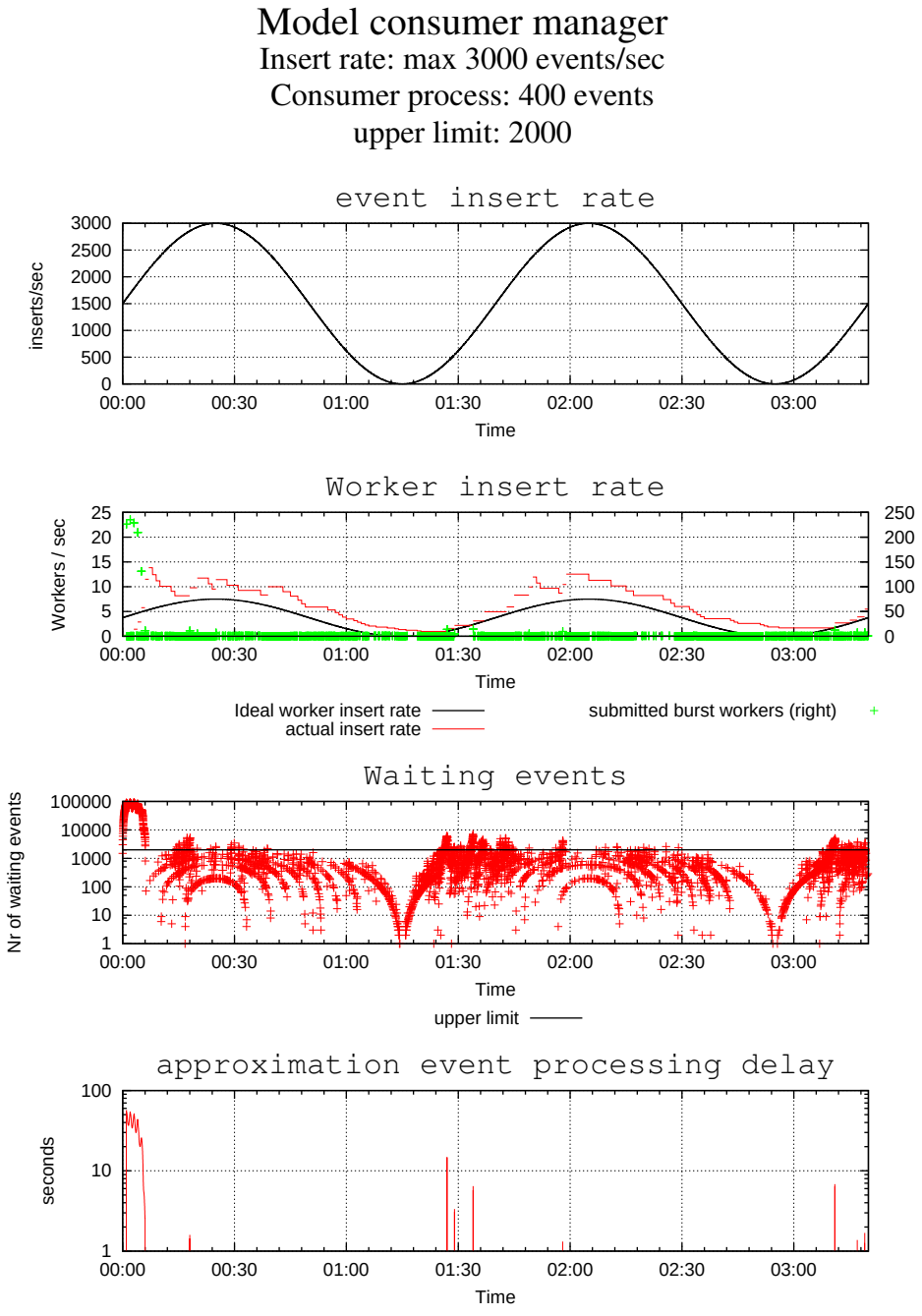

Figure B.40: As B.36, upper limit adjusted to 2000. These are the settings employed in the Proof of Concept solution. Together with B.37, B.38, B.39, B.40, B.41 and B.42 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently. A comparison with the series started by B.29 is hard to make, but waiting events seem to be more spread. A more in depth discussion is held in section 4.2.2.3 on page 44

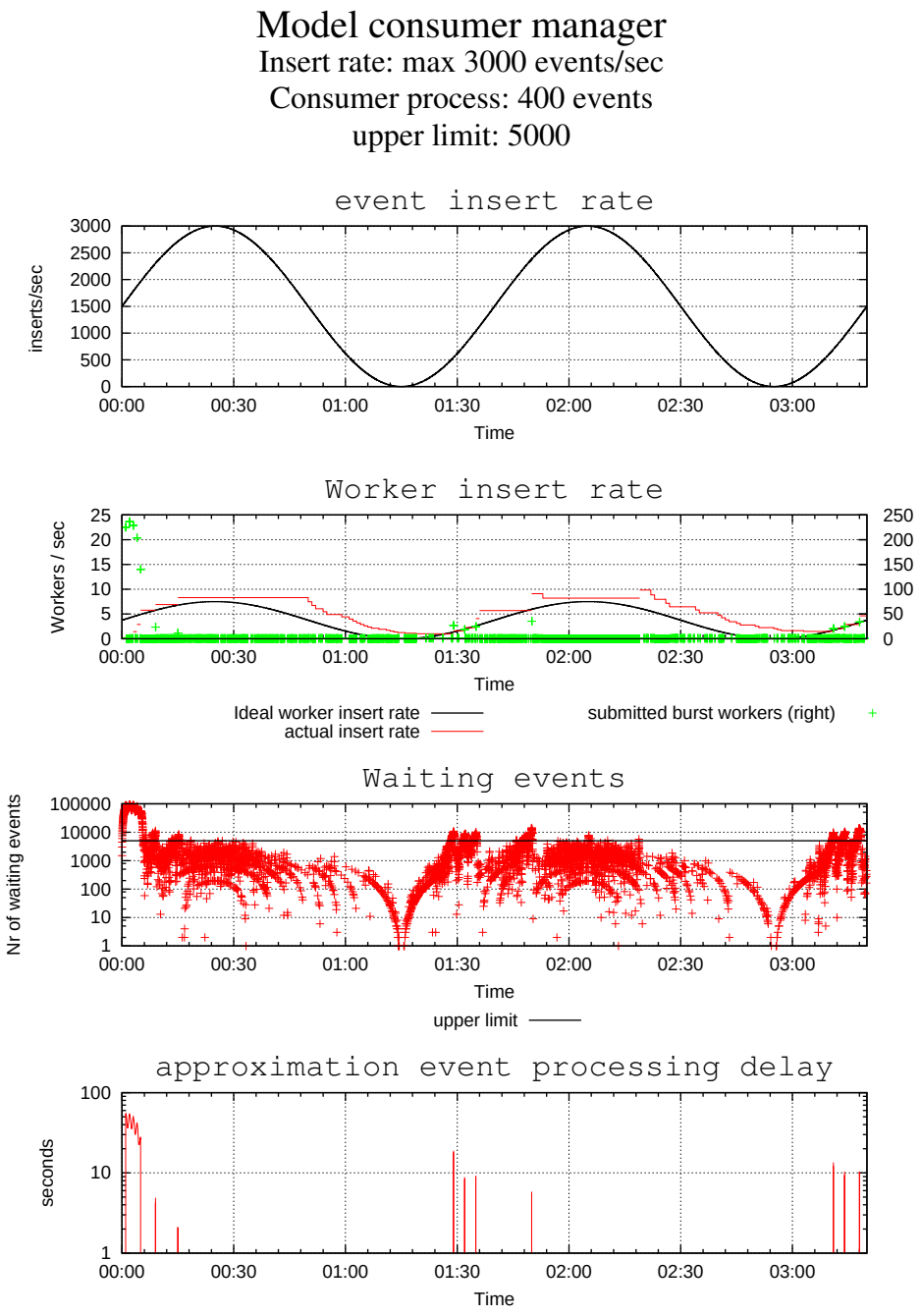

Figure B.41: As B.36, upper limit adjusted to 5000. These are the settings employed in the Proof of Concept solution. Together with B.37, B.38, B.39, B.40, B.41 and B.42 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently. A comparison with the series started by B.29 is hard to make, but waiting events seem to be more spread. A more in depth discussion is held in section 4.2.2.3 on page 44

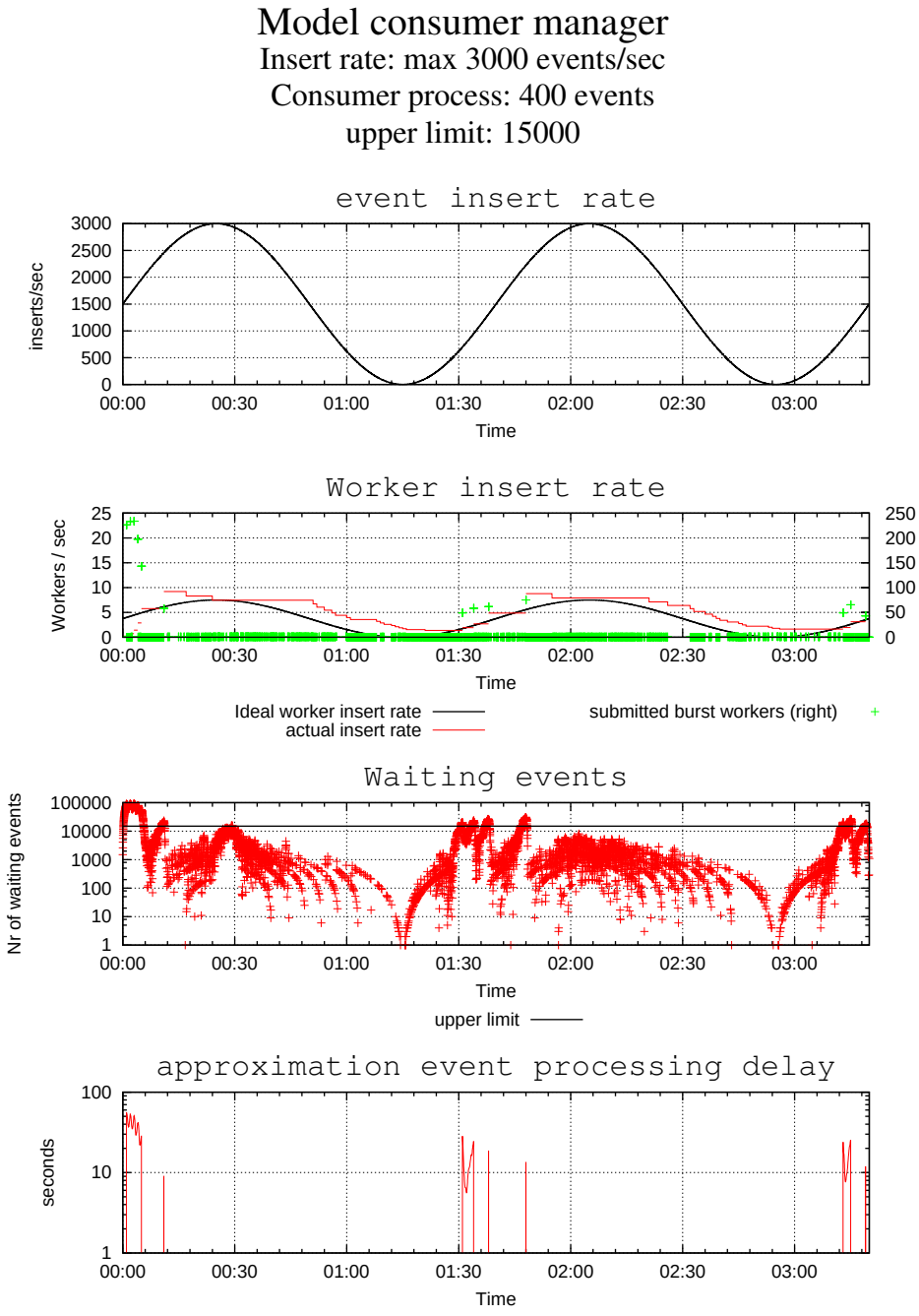

Figure B.42: As B.36, upper limit adjusted to 15000. These are the settings employed in the Proof of Concept solution. Together with B.37, B.38, B.39, B.40, B.41 and B.42 these plots explore the algorithm with different upper limit settings. The larger the upper limit the less workers are run but the longer events have to wait on average. By comparing with the series started by plot B.22 it can be seen that adjustments happen more fluently. A comparison with the series started by B.29 is hard to make, but waiting events seem to be more spread. A more in depth discussion is held in section 4.2.2.3 on page 44

B.2.2 PoC test run

This section contains the plots created during the normal test run as explained in the enviroment of the PoC, see section 4.1.3.2 on page 38 and section 4.2.3.2 on page 46.

All plots are grouped by container and then by host, and are described by there caption.

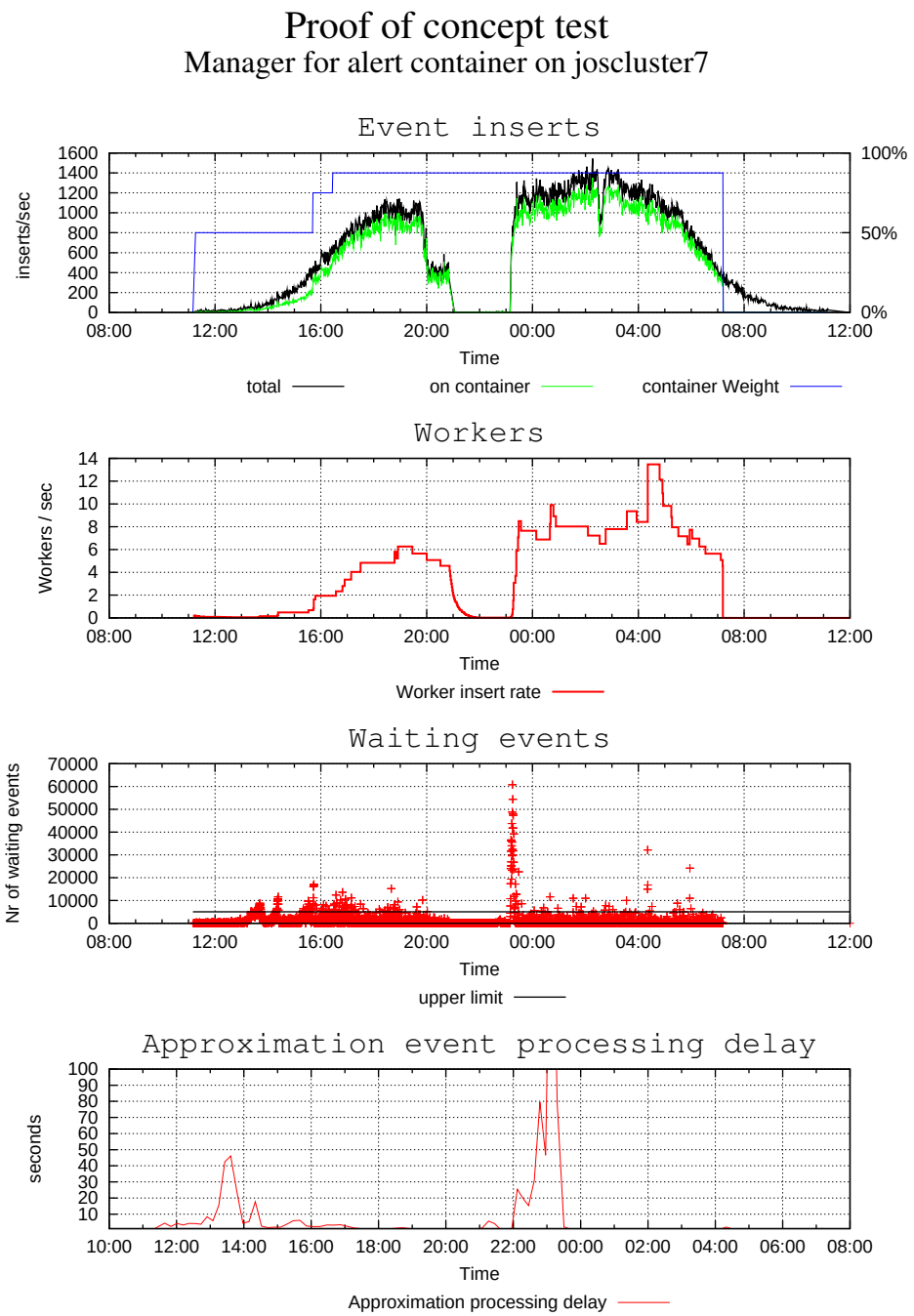

Figure B.43: This plot shows the behavior of the manager for the alert container on joscluster7. Together with B.44 and B.45 it shows the total processing capacity for the alert queue. Around 08:00 it can be seen that the queue system decided to offload the container to joscluster8 (B.44 and B.10) the manager is therefore stopped. Please note that the event processing delay is an approximation and unreliable with low insert rates.

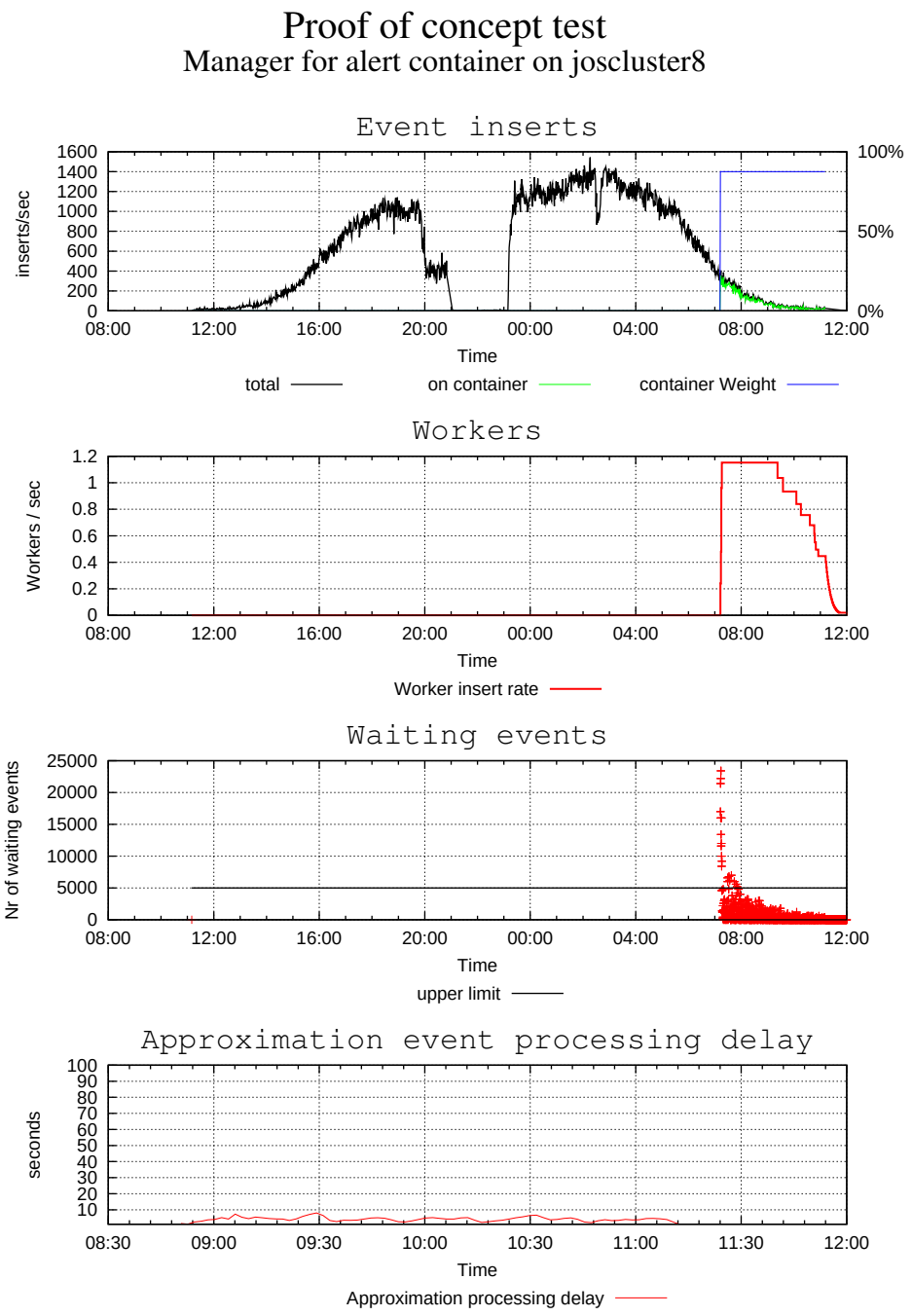

Figure B.44: This plot shows the behavior of the manager for the alert container on joscluster8. Together with B.43 and B.45 it shows the total processing capacity for the alert queue. Around 08:00 it can be seen that the queue system decided to offload the alert container on joscluster7 to joscluster8 (B.43 and B.10) the manager is therefore started.

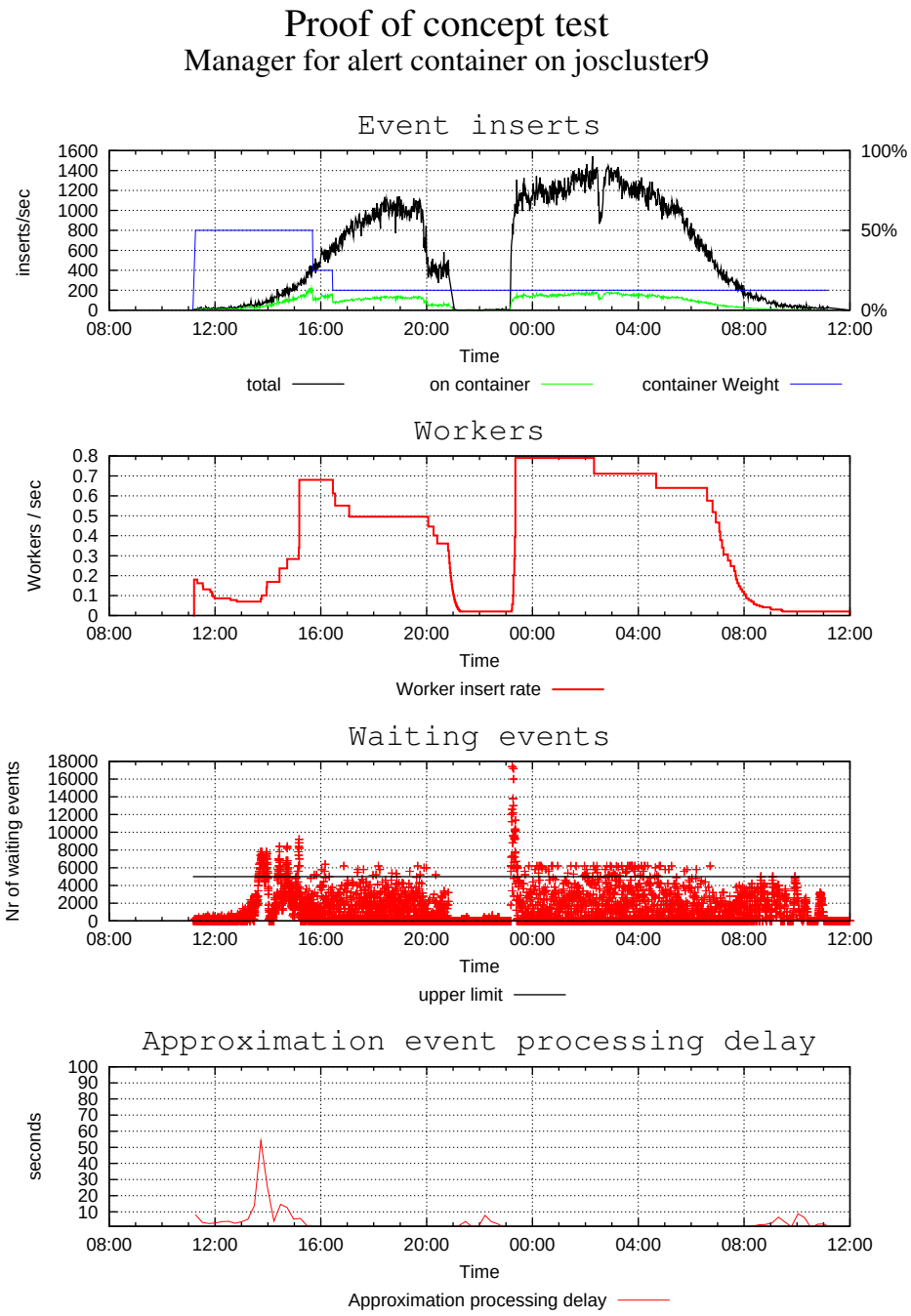

Figure B.45: This plot shows the behavior of the manager for the alert container on joscluster9. Together with B.43 and B.44 it shows the total processing capacity for the alert queue. Around 16:00 it can be seen that joscluster9 becomes TOOBUSY and offloads wait to joscluster7 (B.43 and B.10), because the insert rate was rising, there is no real change in rate of events the container receives.

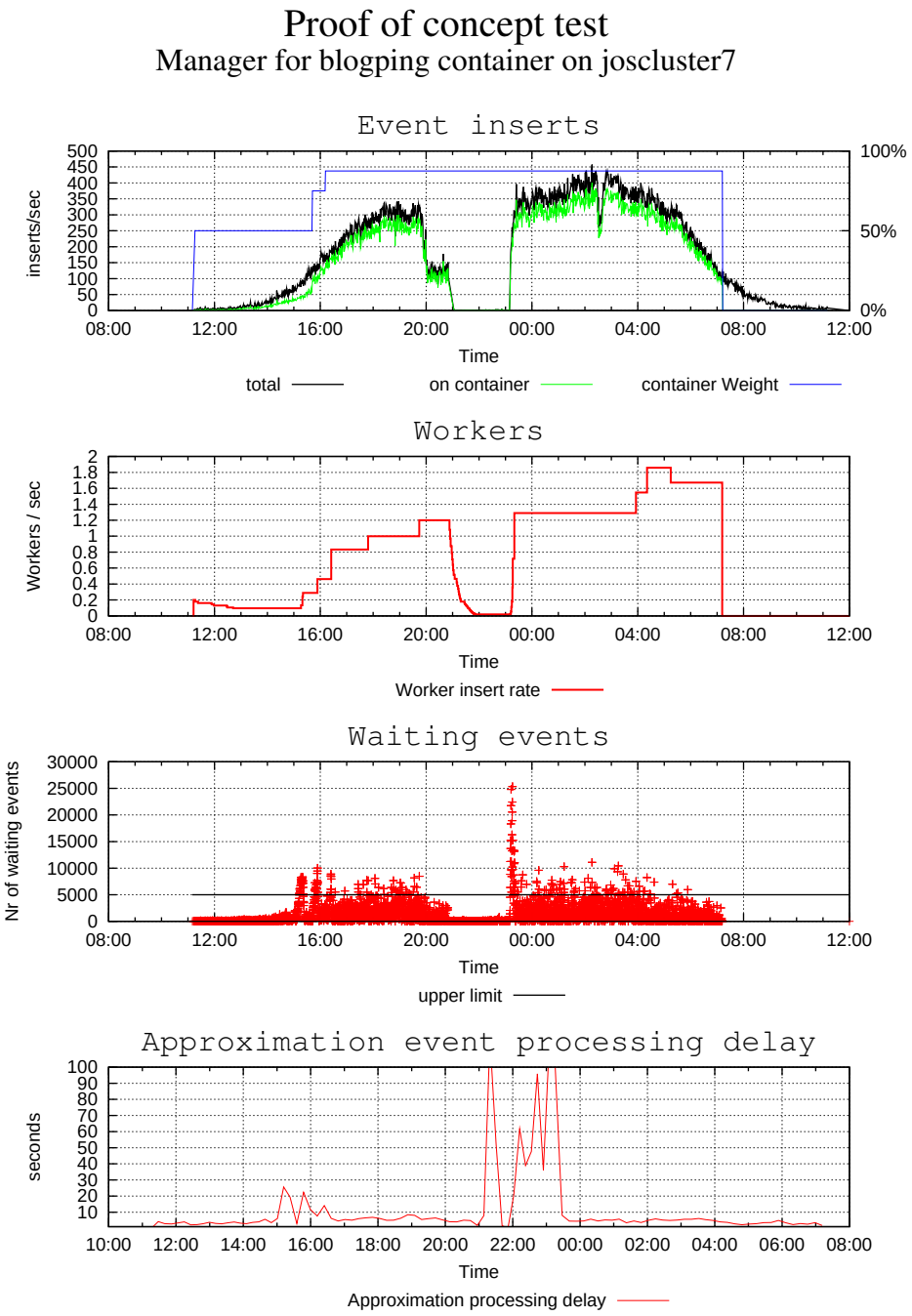

Figure B.46: This plot shows the behavior of the manager for the blogping container on joscluster7. Together with B.47 and B.48 it shows the total processing capacity for the blogping queue. Around 08:00 it can be seen that the queue system decided to offload the container to joscluster8 (B.47 and B.10) the manager is therefore stopped. Please note that the event processing delay is an approximation and unreliable with low insert rates.

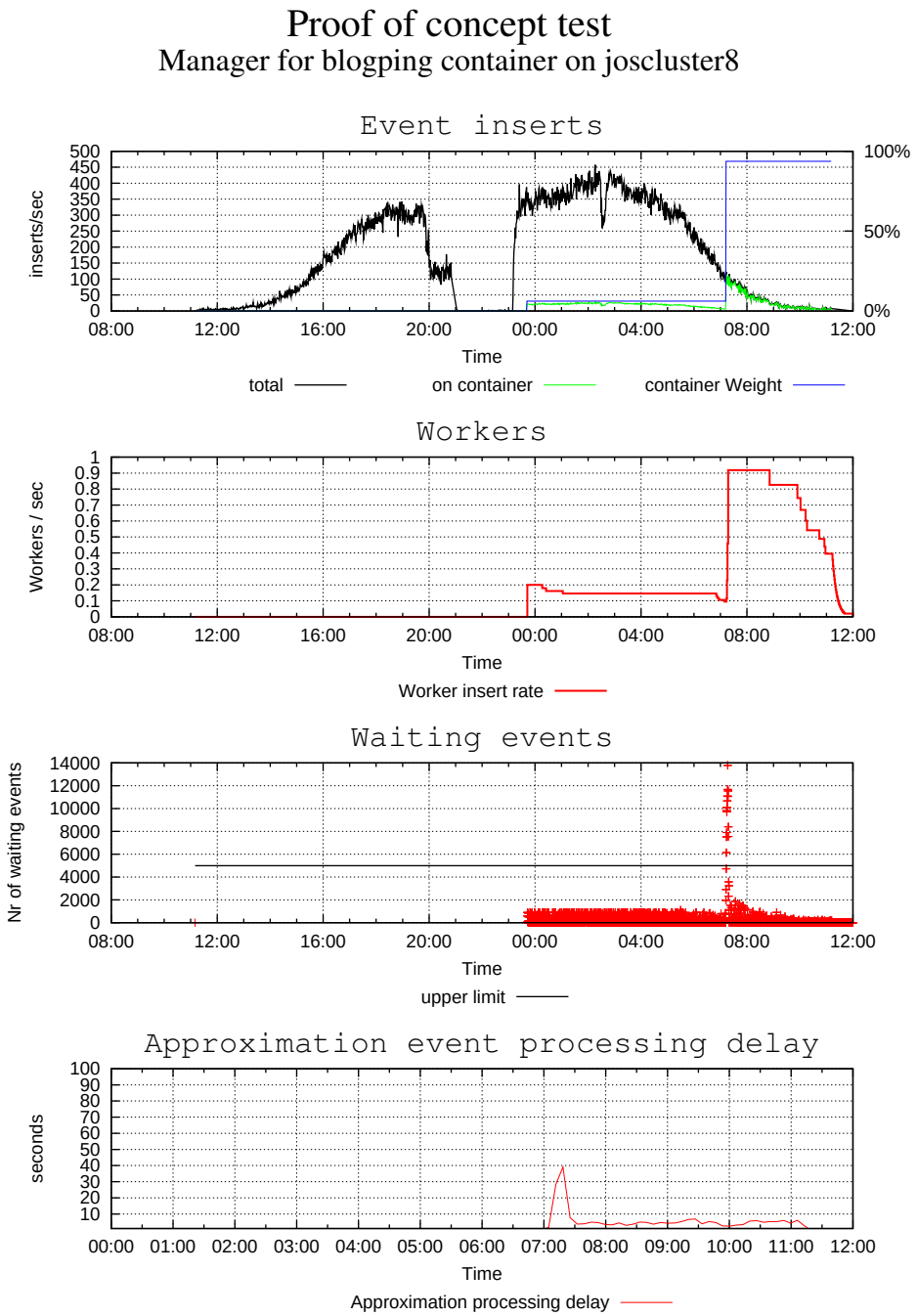

Figure B.47: This plot shows the behavior of the manager for the blogping container on joscluster8. Together with B.46 and B.48 it shows the total processing capacity for the blogping queue. Around 08:00 it can be seen that the queue system decided to offload the blogping container on joscluster7 to joscluster8 (B.46 and B.10) this caused the worker insert rate to jump in order to keep up.

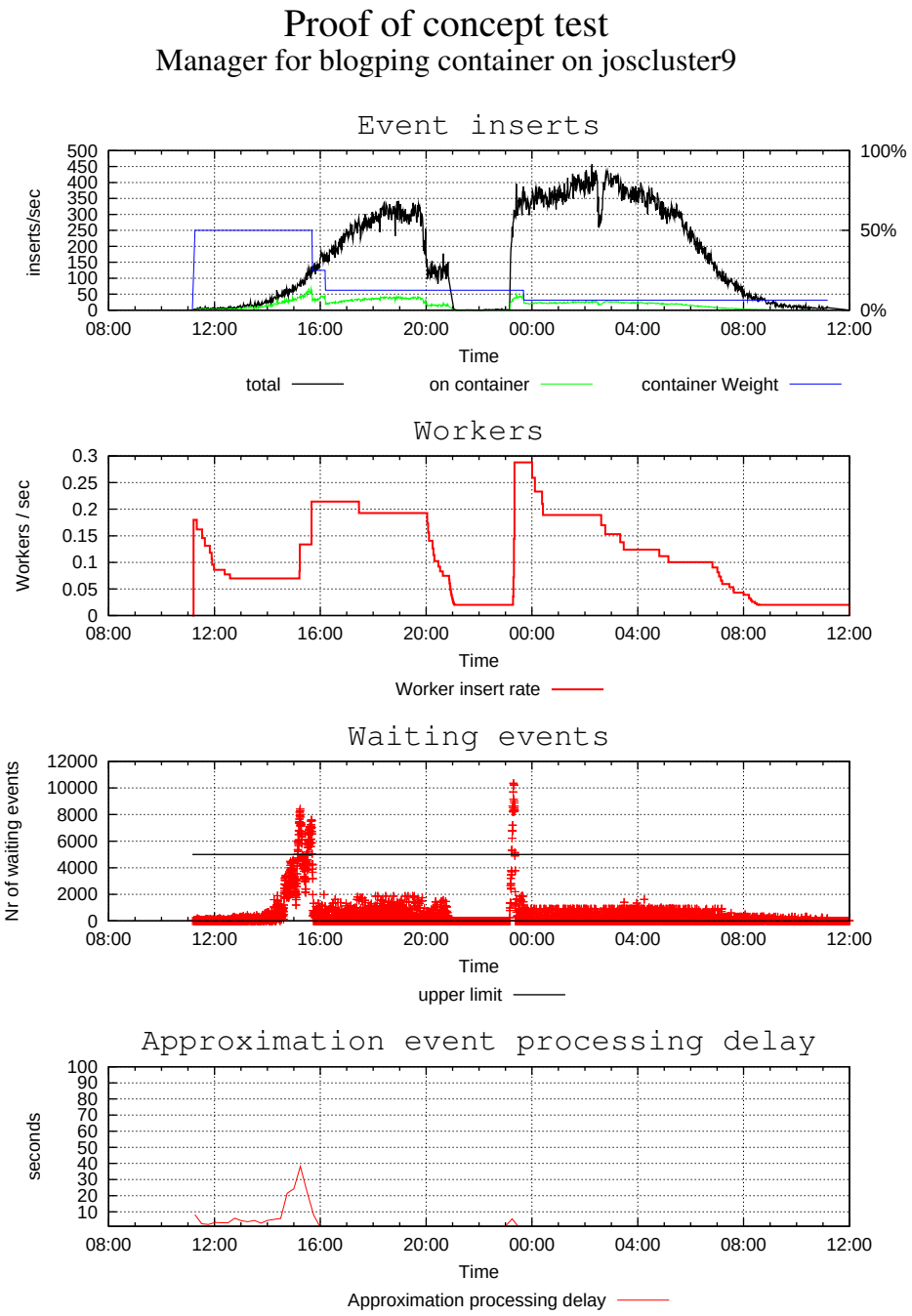

Figure B.48: This plot shows the behavior of the manager for the blogping container on joscluster9. Together with B.46 and B.47 it shows the total processing capacity for the blogping queue. Around 16:00 it can be seen that joscluster9 becomes TOOBUSY and offloads the container to joscluster7 (B.46 and B.10), because the insert rate was rising, there is no real change in rate of events the container receives.

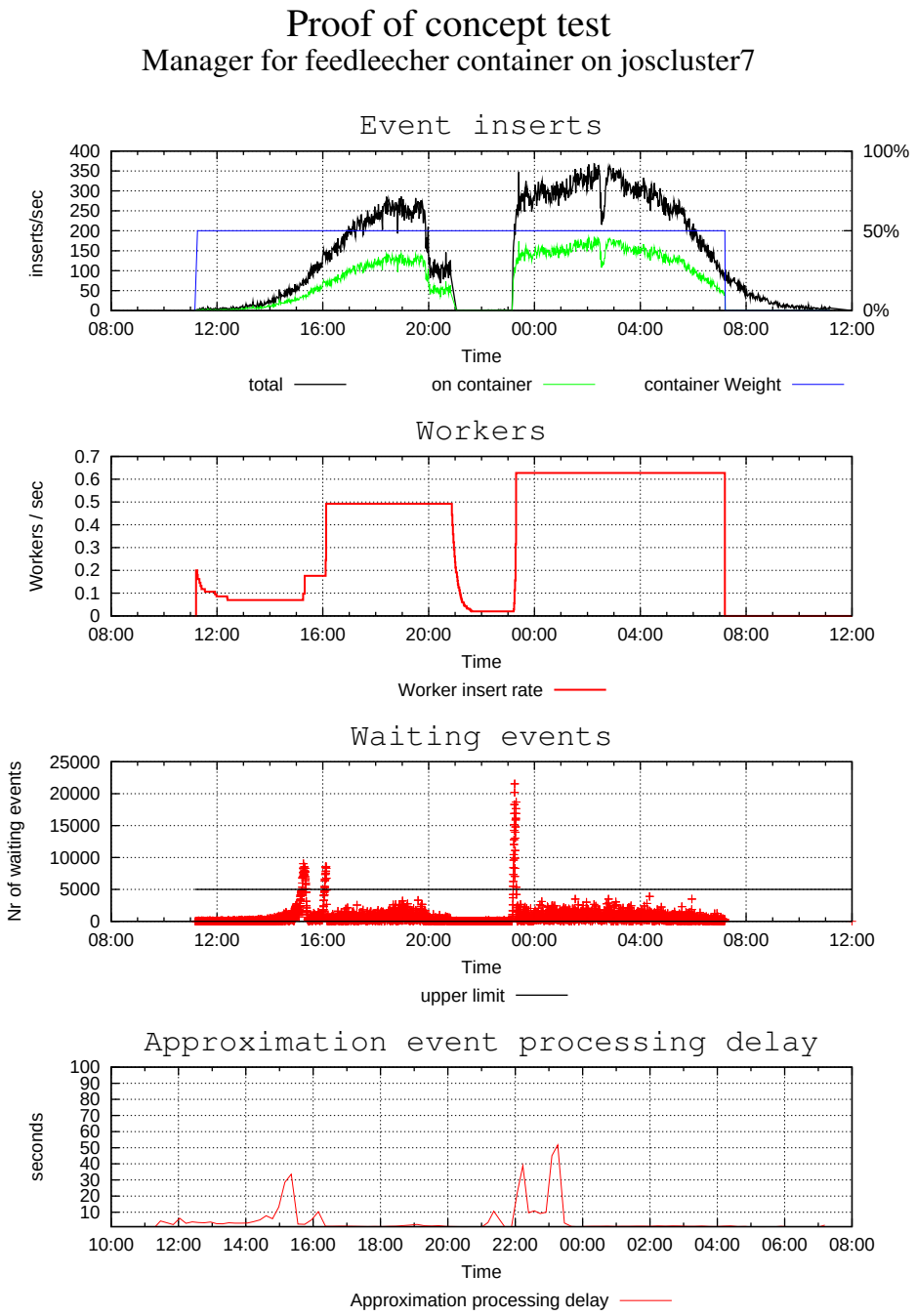

Figure B.49: This plot shows the behavior of the manager for the feedleecher container on joscluster7. Together with B.50 and B.51 it shows the total processing capacity for the feedleecher queue. Around 08:00 it can be seen that the queue system decided to offload the container to joscluster8 (B.50 and B.10) the manager is therefore stopped. Please note that the event processing delay is an approximation and unreliable with low insert rates.

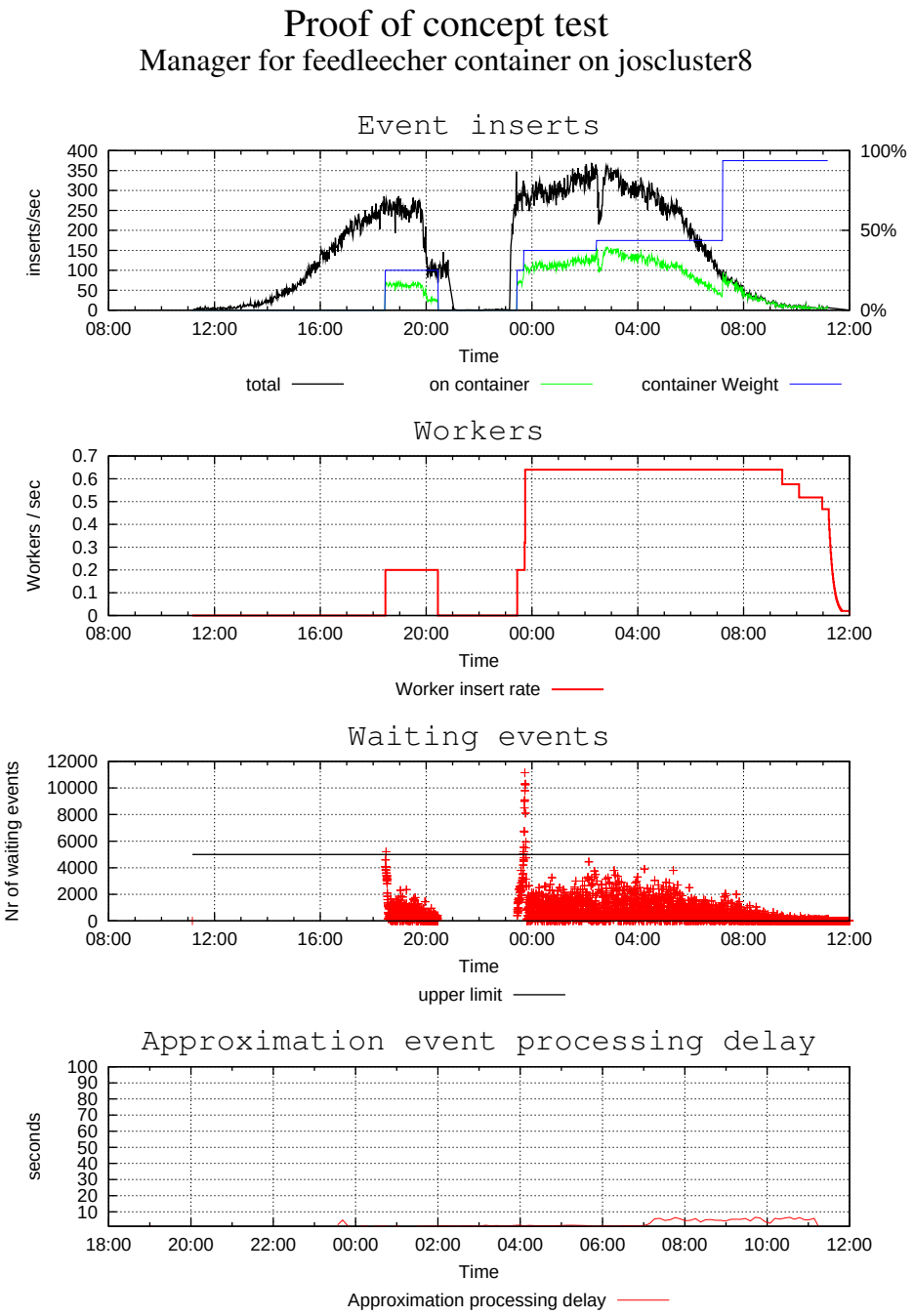

Figure B.50: This plot shows the behavior of the manager for the feedleecher container on joscluster8. Together with B.49 and B.51 it shows the total processing capacity for the feedleecher queue. Around 08:00 it can be seen that the queue system decided to offload the feedleecher container on joscluster7 to joscluster8 (B.49 and B.10) due to the declining insert rates this has no effect on the workers.

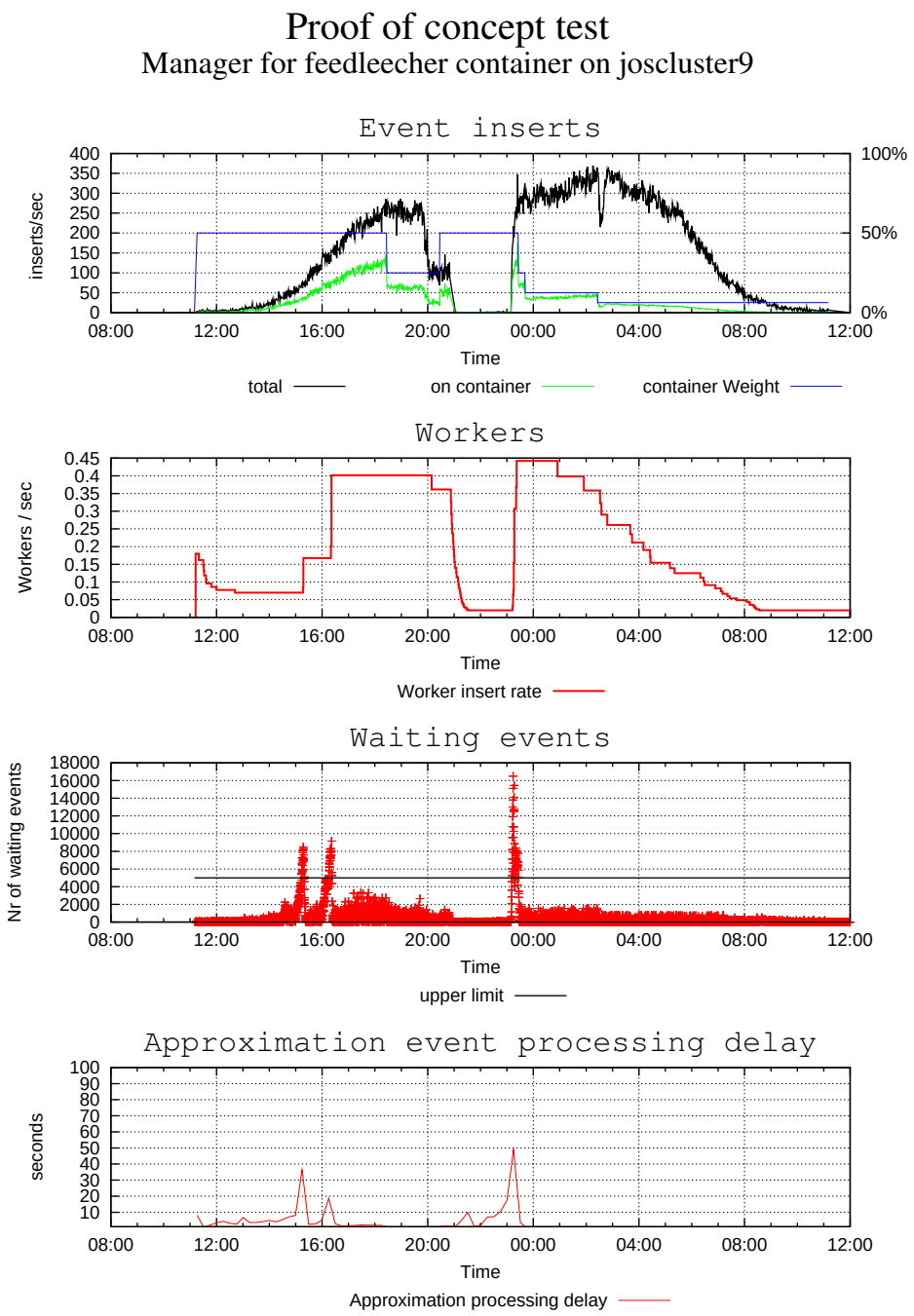

Figure B.51: This plot shows the behavior of the manager for the feedleecher container on joscluster9. Together with B.49 and B.50 it shows the total processing capacity for the feedleecher queue.

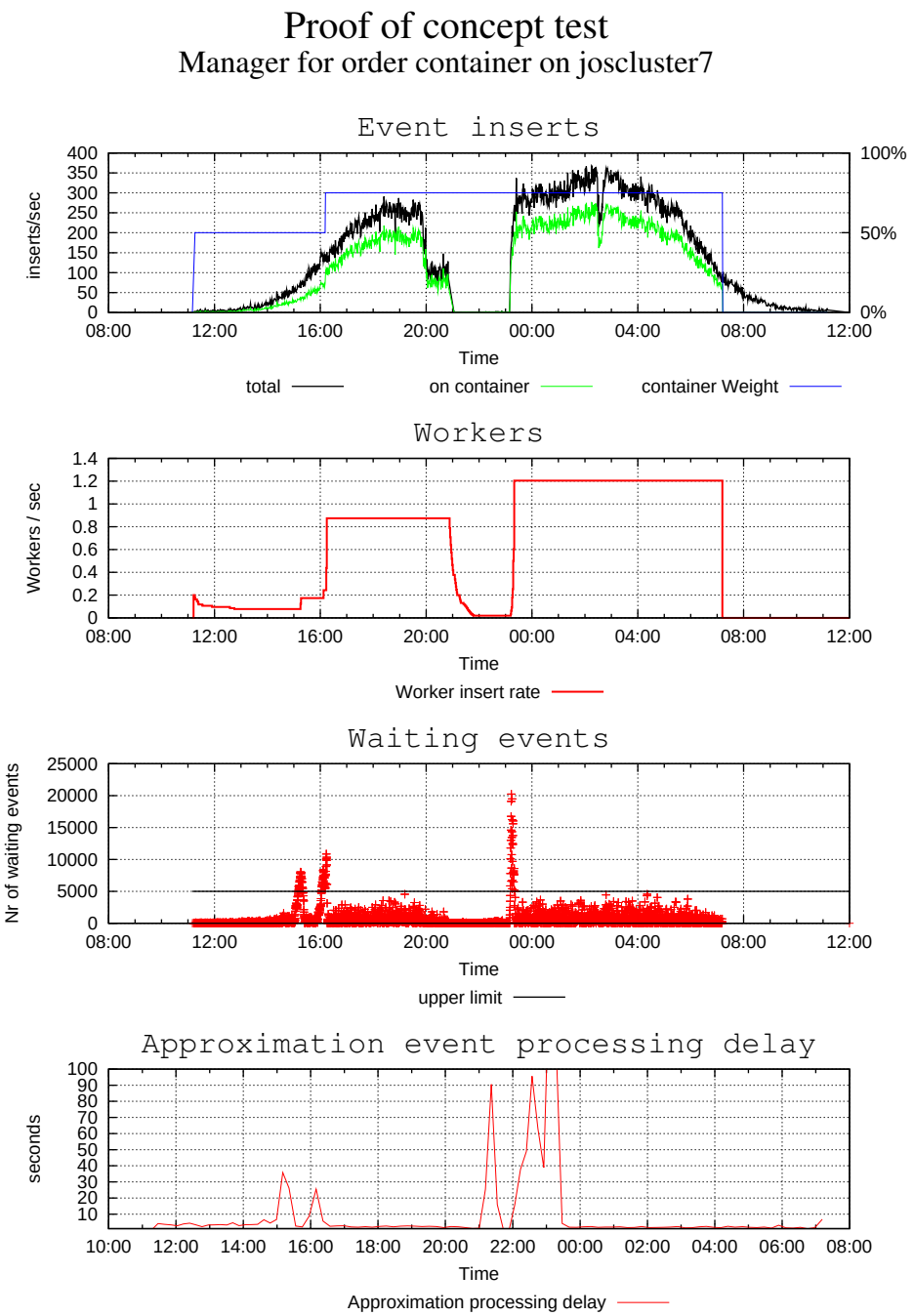

Figure B.52: This plot shows the behavior of the manager for the order container on joscluster7. Together with B.53 and B.54 it shows the total processing capacity for the order queue. Around 08:00 it can be seen that the queue system decided to offload the container to joscluster8 (B.53 and B.10) the manager is therefore stopped. Please note that the event processing delay is an approximation and unreliable with low insert rates.

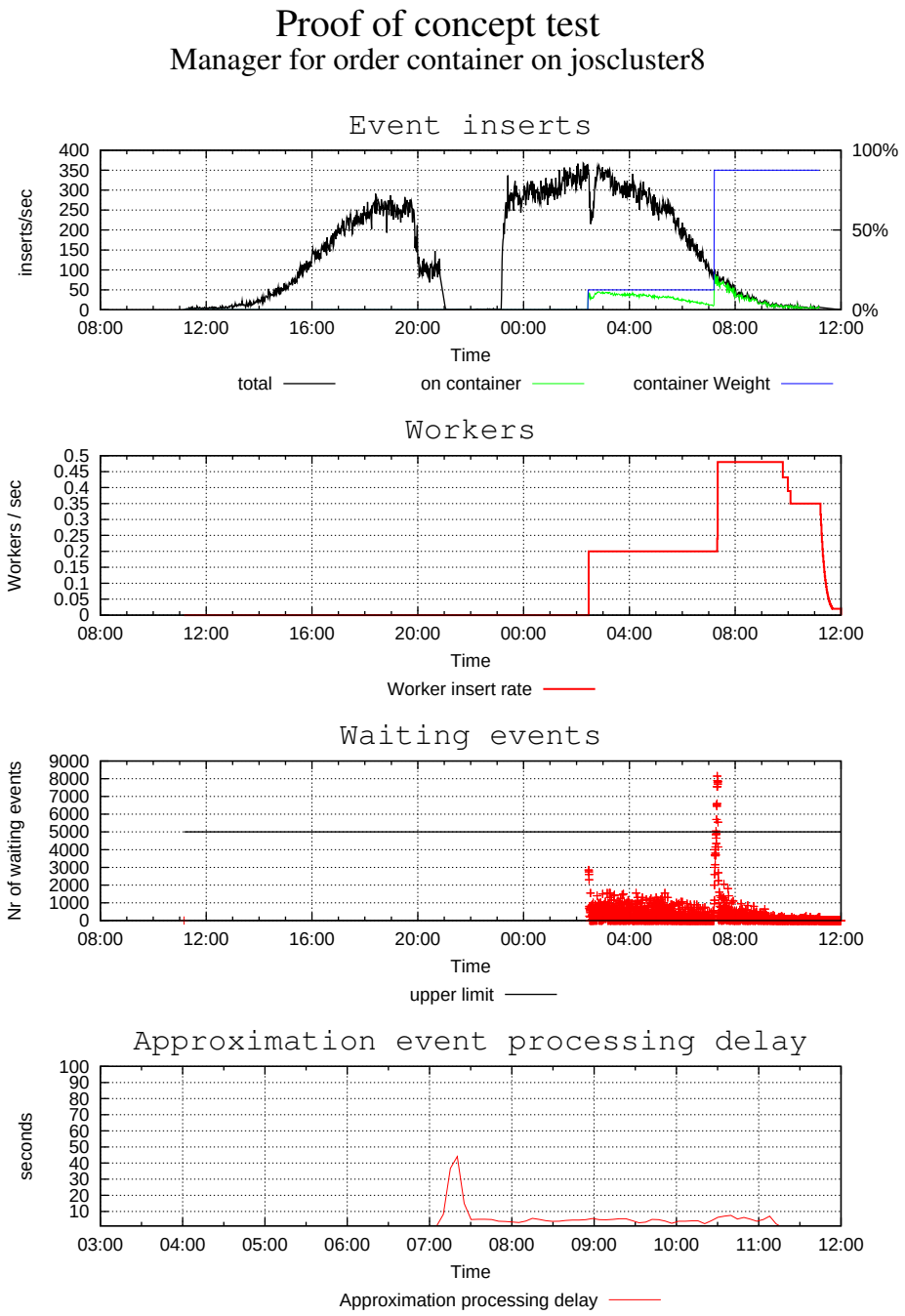

Figure B.53: This plot shows the behavior of the manager for the order container on joscluster8. Together with B.52 and B.54 it shows the total processing capacity for the order queue. Around 08:00 it can be seen that the queue system decided to offload the order container on joscluster7 to joscluster8 (B.52 and B.10) due to the declining insert rates this has no effect on the workers.

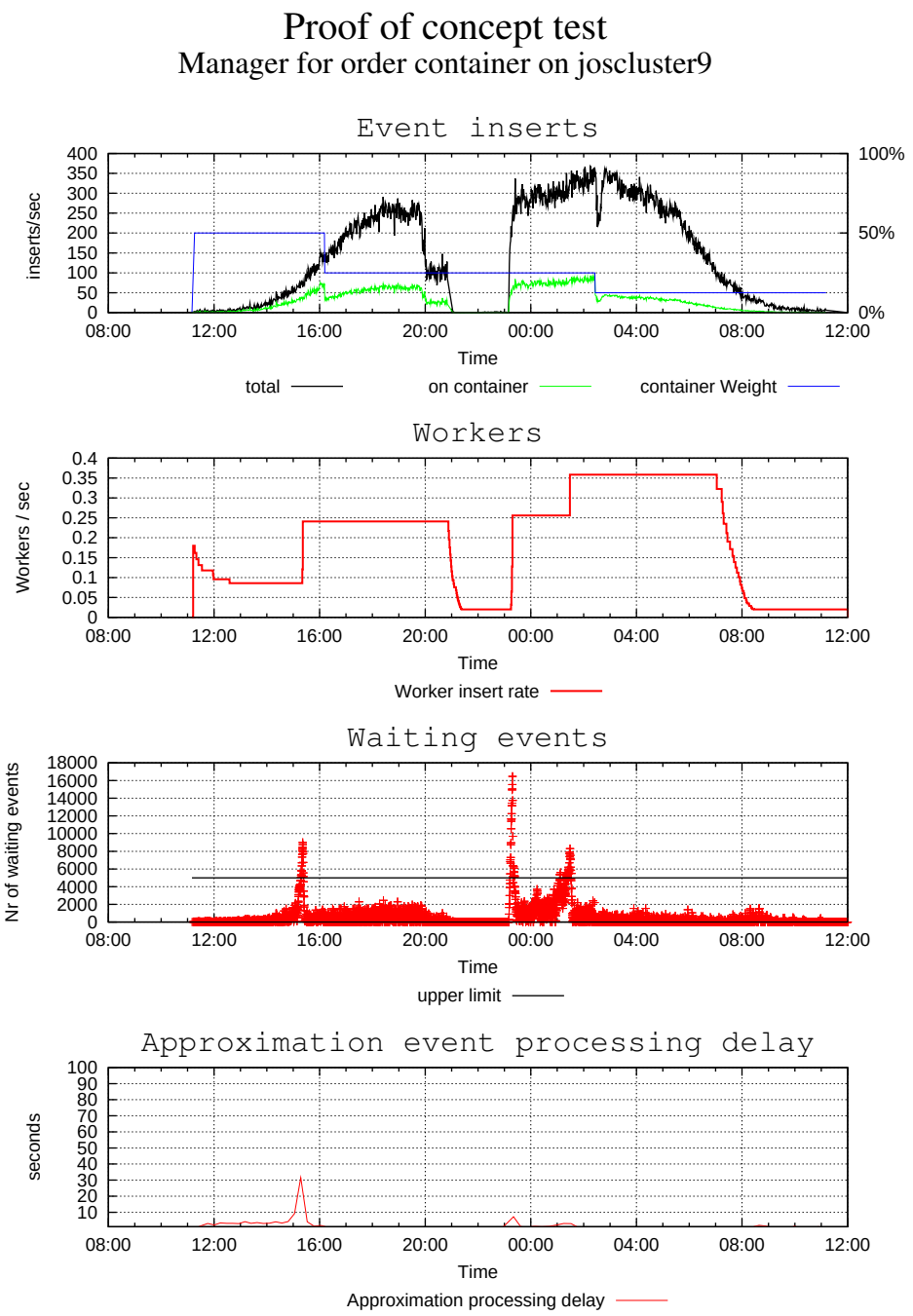

Figure B.54: This plot shows the behavior of the manager for the order container on joscluster9. Together with B.52 and B.53 it shows the total processing capacity for the order queue.

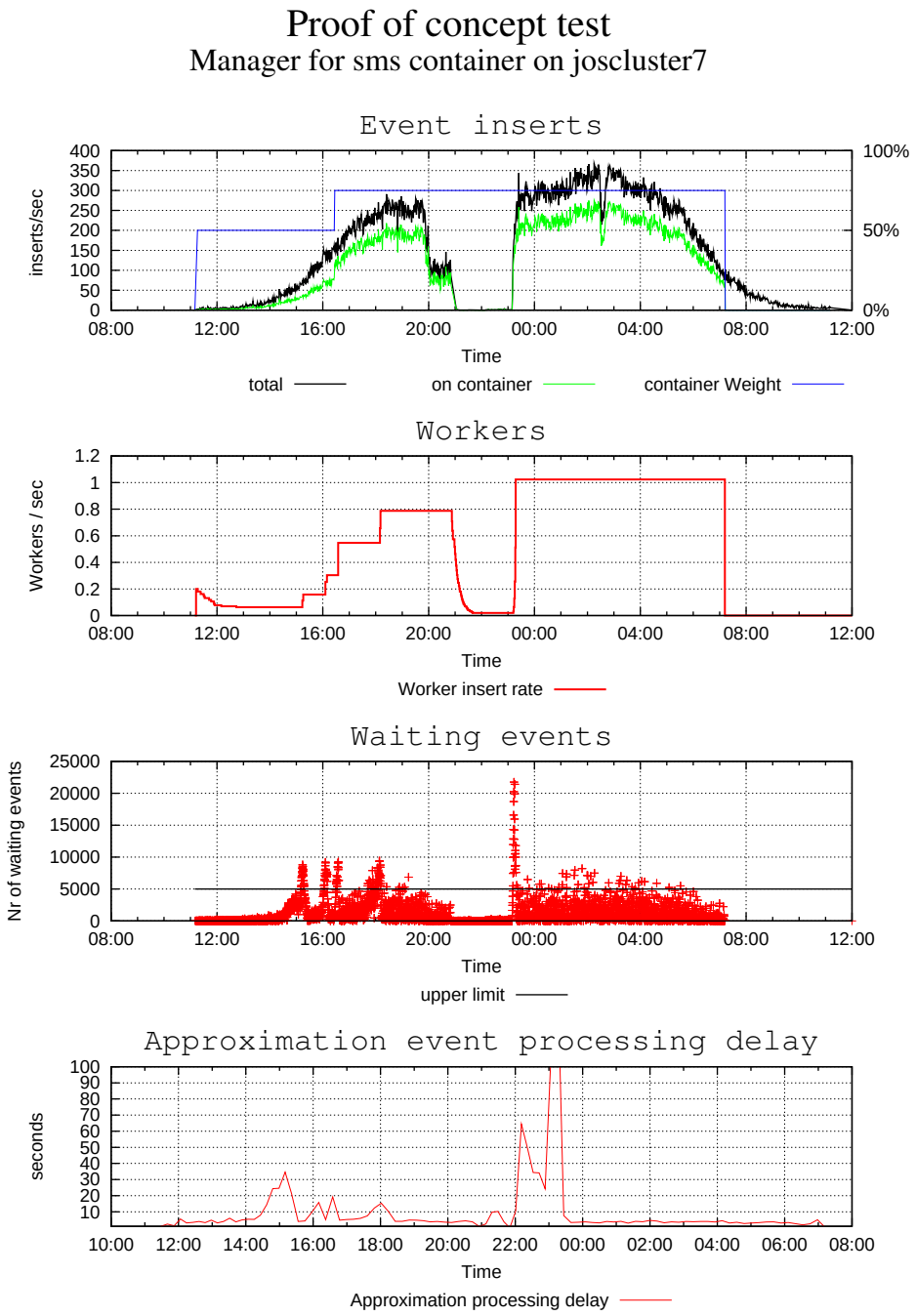

Figure B.55: This plot shows the behavior of the manager for the sms container on joscluster7. Together with B.56 and B.57 it shows the total processing capacity for the sms queue. Around 08:00 it can be seen that the queue system decided to offload the container to joscluster8 (B.56 and B.10) the manager is therefore stopped. Please note that the event processing delay is an approximation and unreliable with low insert rates.

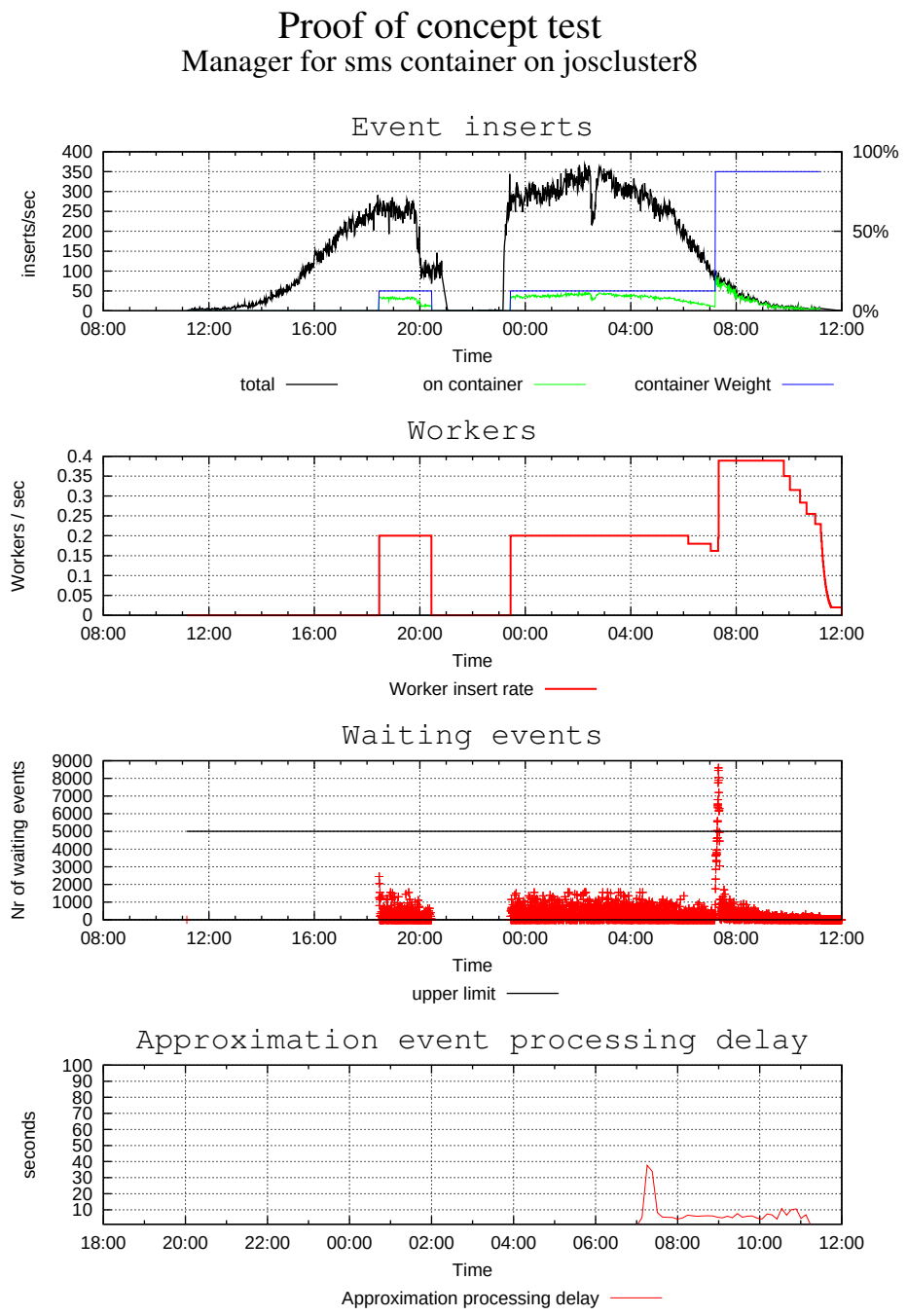

Figure B.56: This plot shows the behavior of the manager for the sms container on joscluster8. Together with B.55 and B.57 it shows the total processing capacity for the sms queue. Around 08:00 it can be seen that the queue system decided to offload the sms container on joscluster7 to joscluster8 (B.55 and B.10).

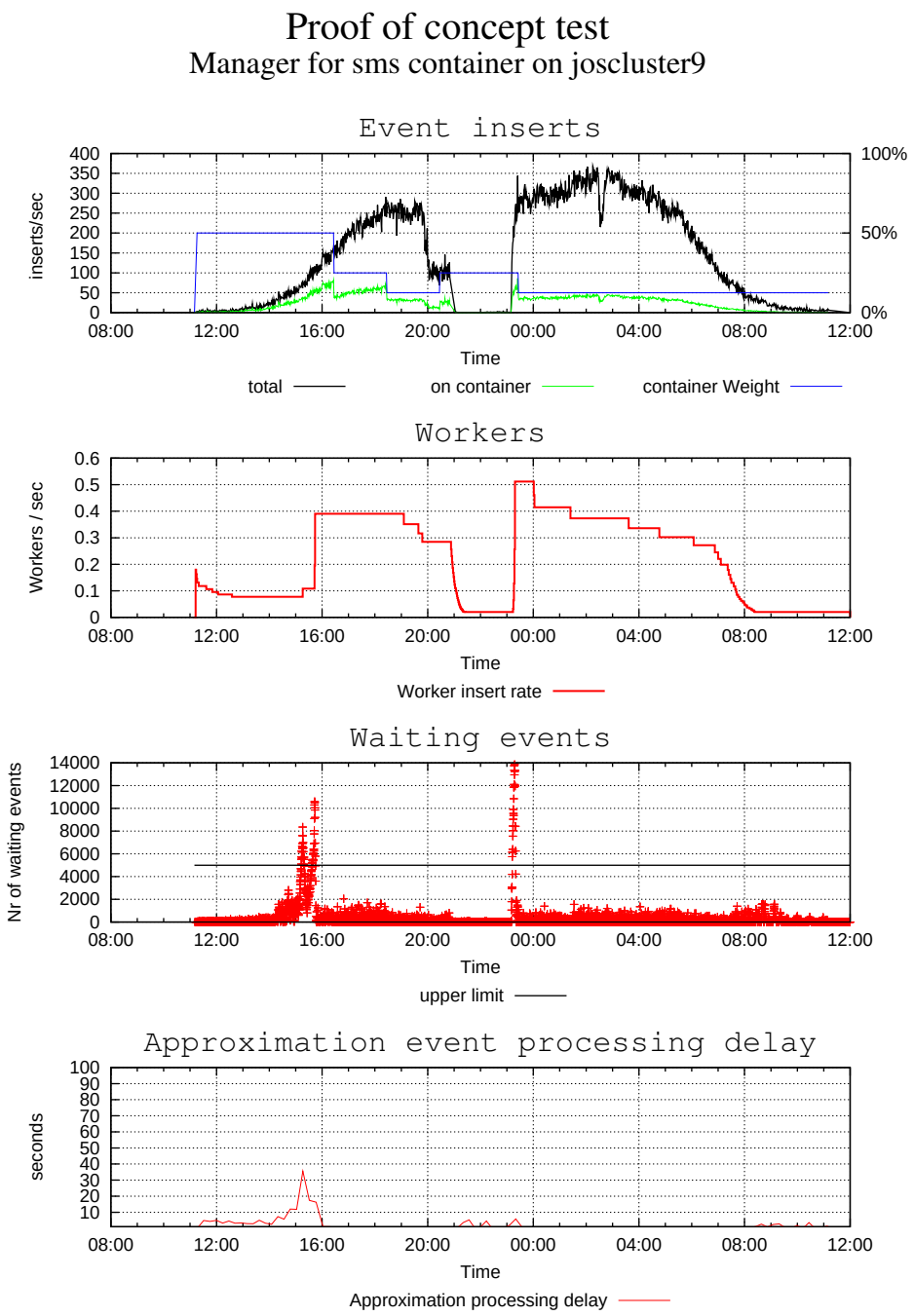

Figure B.57: This plot shows the behavior of the manager for the sms container on joscluster9. Together with B.55 and B.56 it shows the total processing capacity for the sms queue.

B.2.3 Standalone tests

B.2.3.1 Overload

This section contains the plots for the queue system overload test, discussed in section 4.2.3.3 on page 47.

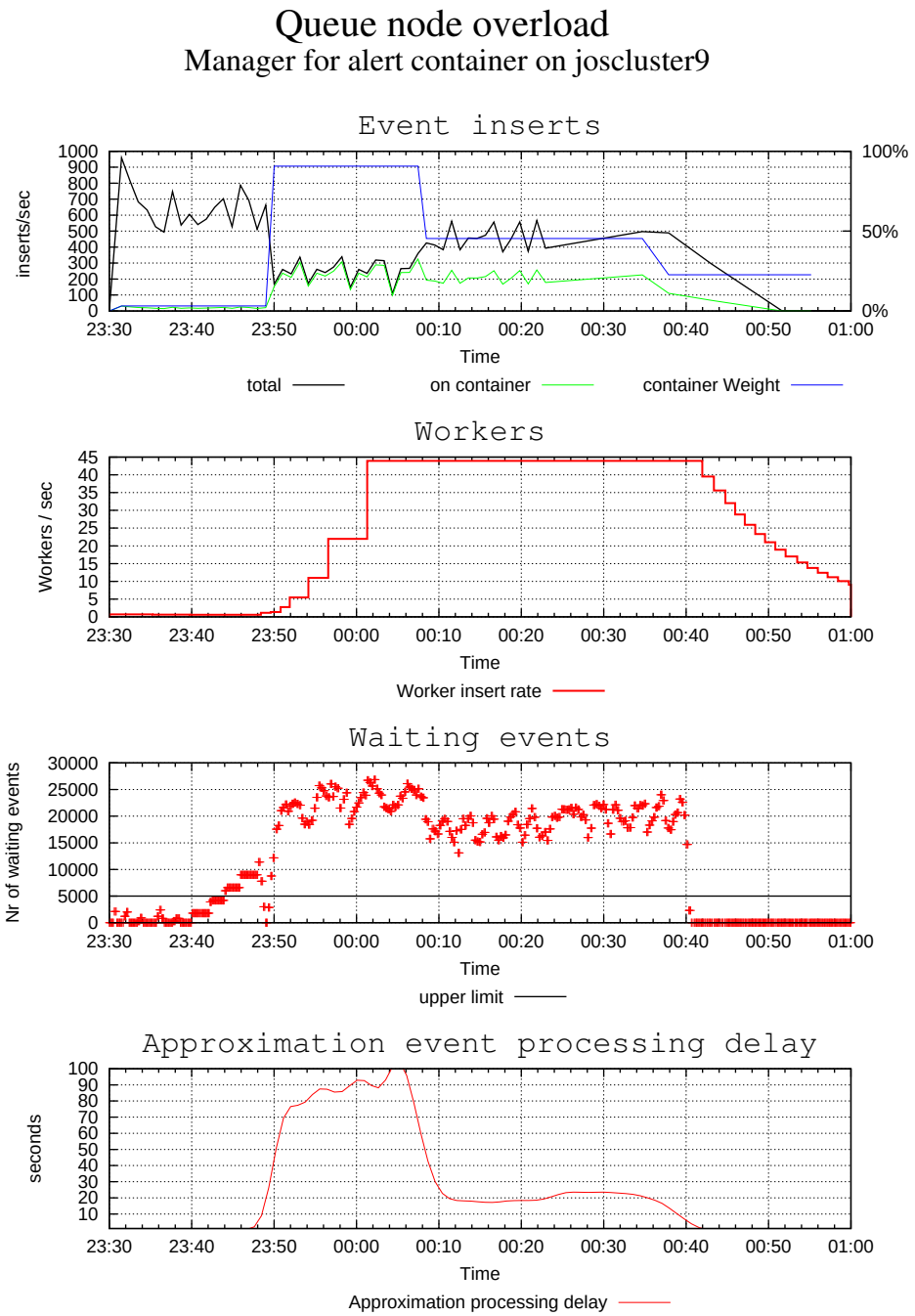

Figure B.58: This plot shows the behavior of the manager for the sms container on joscluster9 during an overload. B.59 shows that joscluster9 becomes overloaded when joscluster7 offloads containers to it, in order to downscale the system. This results in slow processing of events and thus the number of waiting events rises. The manager reacts by raising the worker insert rate, further stressing joscluster9. The test is aborted around 00:40.

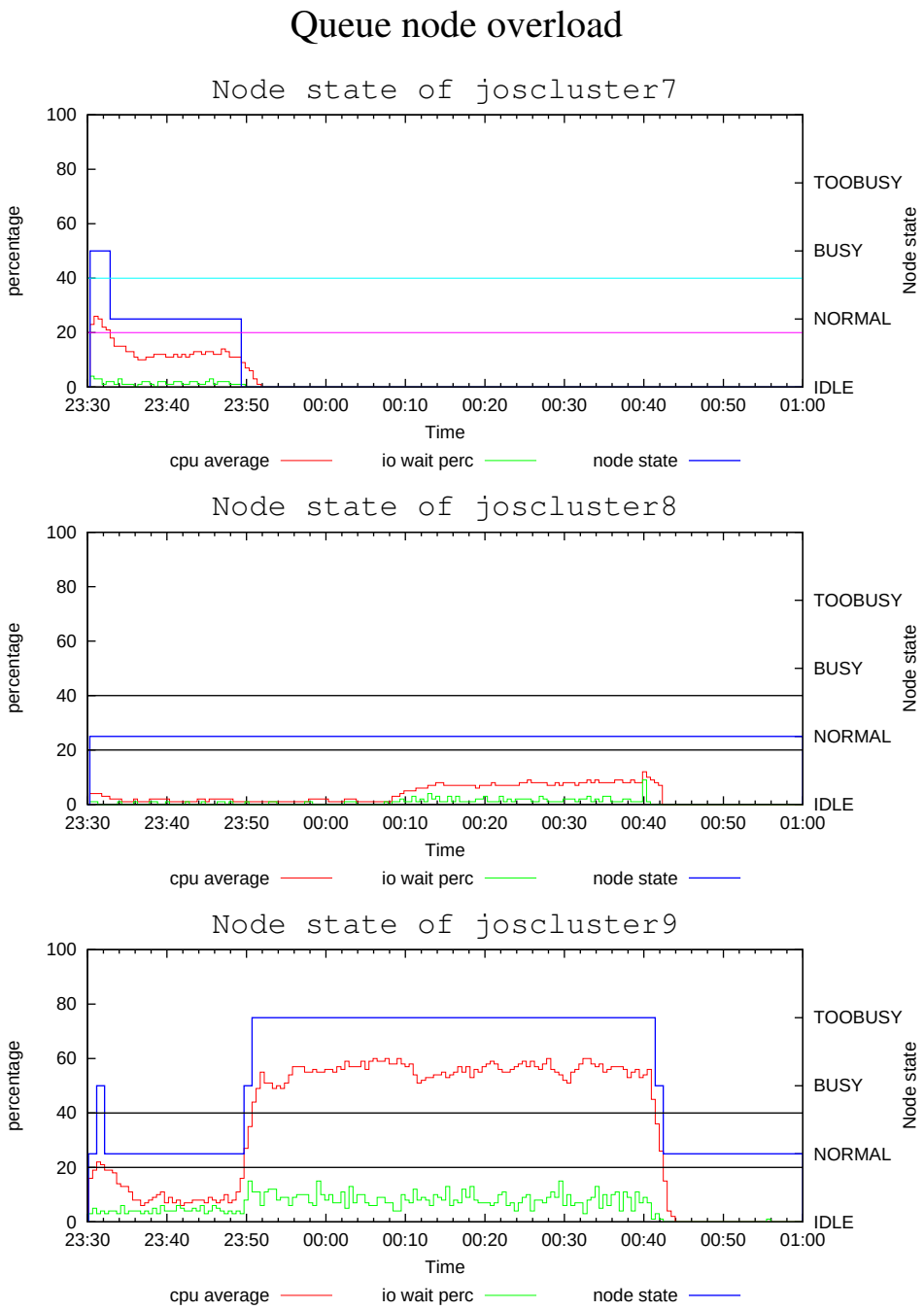

Figure B.59: Cpu and io resource usage during the overload, node state is alse plotted.

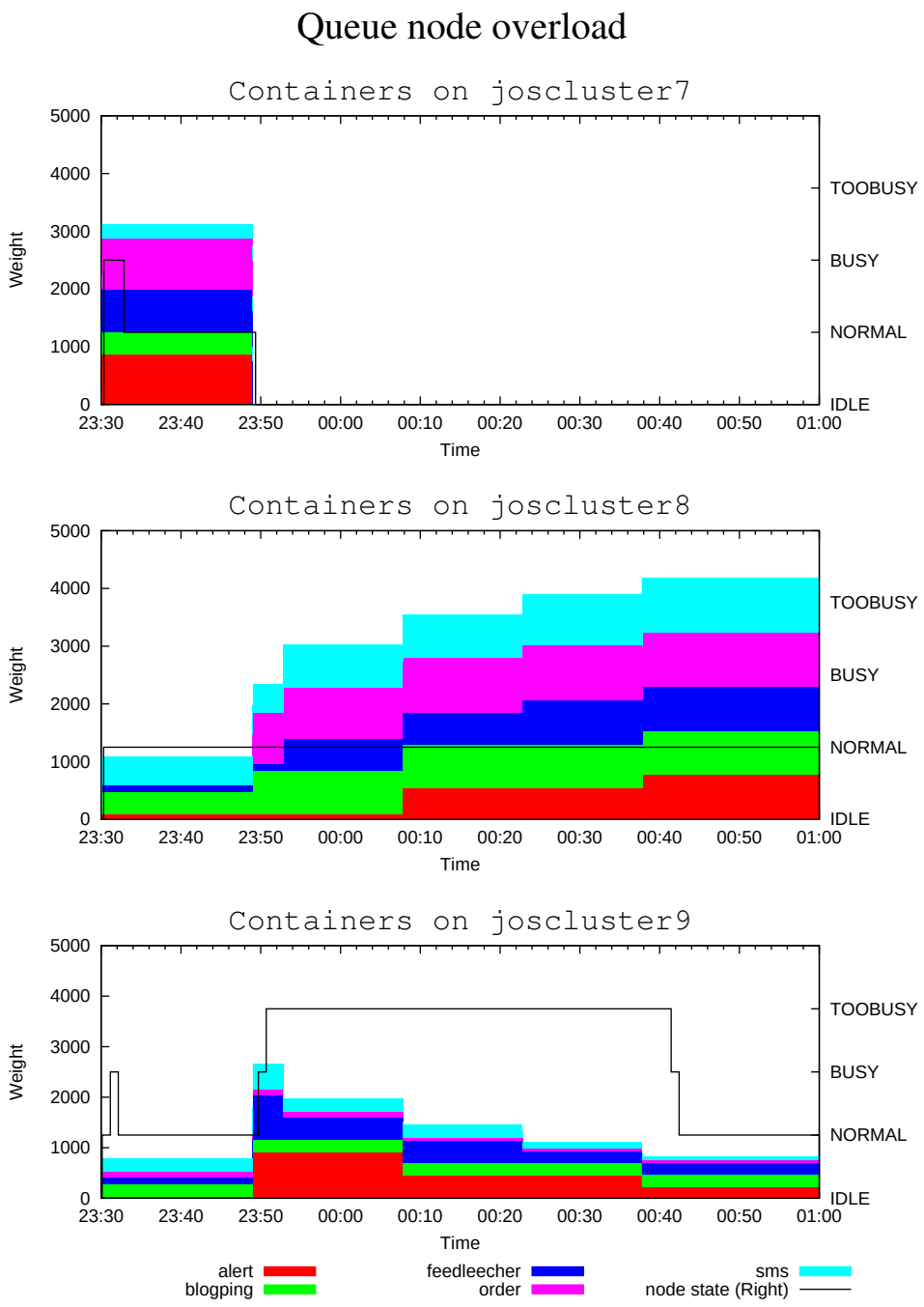

Figure B.60: The container distribution during the overload. The plot shows that joscluster9 tries to move part of its containers every 15 minutes in an attempt to lower the load. The combination of non optimal choices made and the manager which raised its worker insert rate ensures that the situation is not remedied.

B.2.3.2 Backlog

This section contains the plots for the backlog test, discussed in section 4.2.3.3 on page 47.

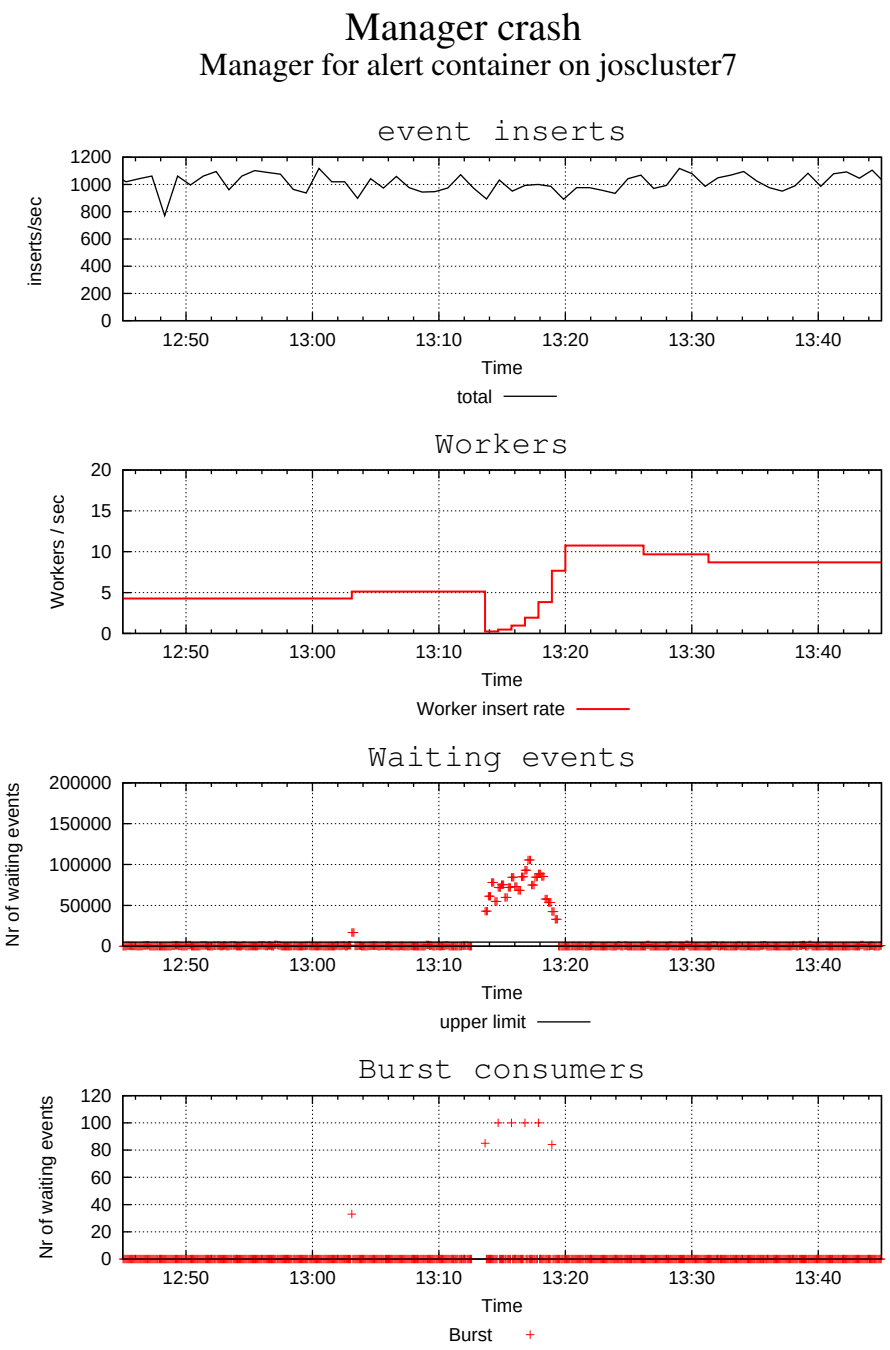

Figure B.61: The plots shows the reaction of a manager when it crashes and is restarted. The manager is forcefully stopped at 13:14, after which the batch system automatically restarts it. The burst consumers plot shows how many extra consumers are started in order to keep up while the worker insert rate is too low.

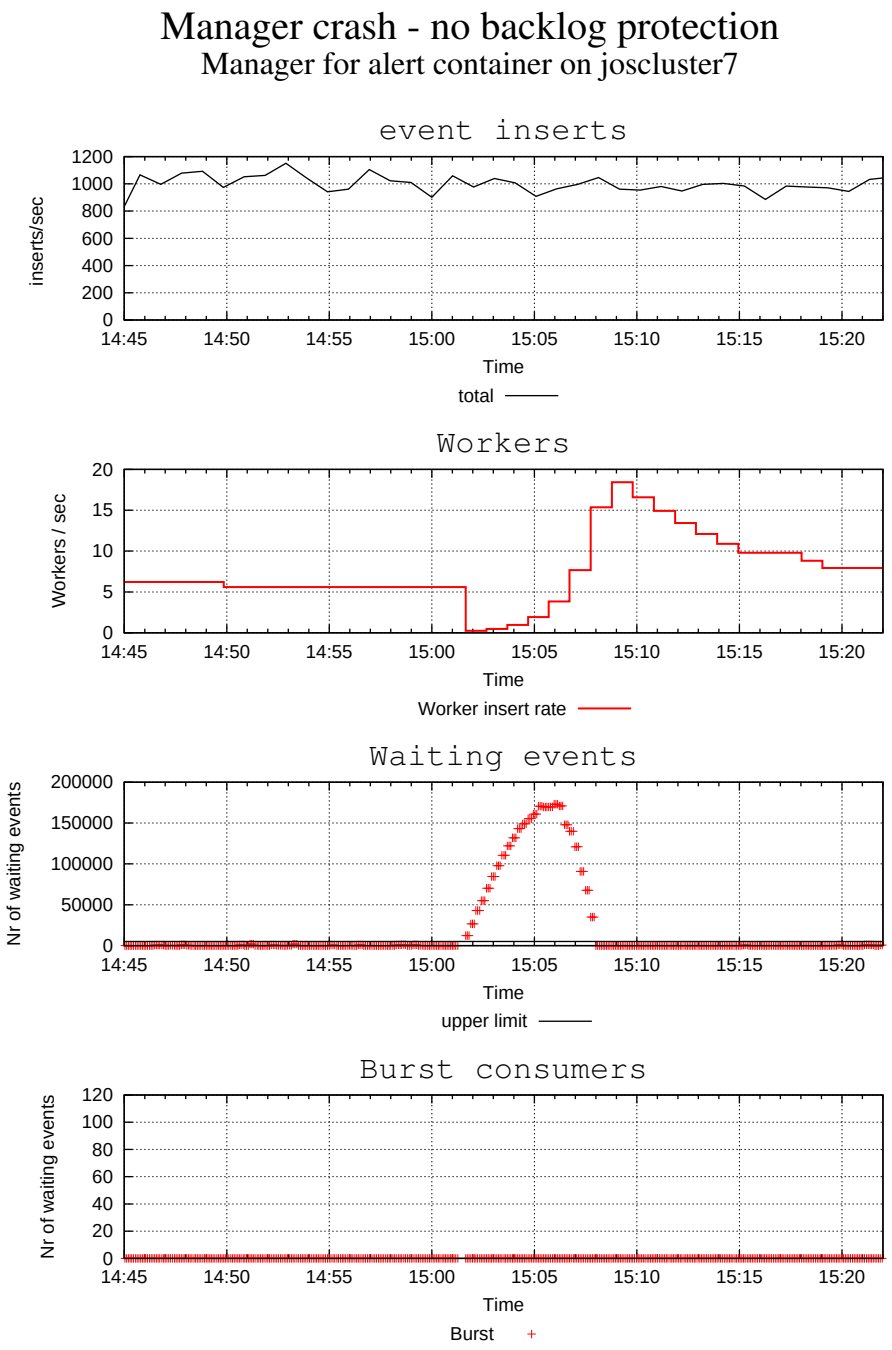

Figure B.62: As in B.61 this plots shows the reaction of a manager in case of crash (which occurred around 15:00), but the backlog mechanism is disabled. This illustrates that without the backlog system the manager will overshoot the ideal worker insert rate by a factor of at least two.

Bibliography

[1] W. S. A. Adams, J. Nicholas. RFC 3973: Protocol independent multicast - dense mode (pim-dm), Mar. 1994. 3.5.1.3

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. Technical report, Technical Report ACM 1-59593-451-0/06/0010. VMware, August 2006. 2.3.1

[3] C. Amza, A. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers, 2005. (document)

[4] C. Amza, A. L. Cox, and W. Zwaenepoel. Scaling and availability for dynamic content web sites, 2002. (document)

[5] F. Christian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989. 3.1.1.2

[6] E. F. Codd. *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. 1.1.2

[7] R. Drummond and O. Babaoglu. Low-cost clock synchronization. *Distributed Computing*, 6:193–203, 1993. 3.1.1.2

[8] Facebook. http://www.facebook.com/. 1

[9] google. http://www.google.com/. 1

[10] S. grid engine. http://gridengine.sunsource.net/. 2.3.2

[11] C. Henderson. Scalable web architectures common patterns and approaches. Presentation. (document), A.4

[12] hi5. http://www.hi5.com/. 1

[13] hyves. http://www.hyves.nl/. 1

[14] IEEE. Ieee standard 802.1d mac bridges, 2004. 3.1.1.1

[15] P. H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. 2.3.1

[16] H. Kopetz and w. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Trans. Comp.*, C-87(8):933–940, August 1987. 3.1.1.2

[17] T. Lindholm and F. Yellin. *The JavaTM Virtual Machine Specification, Second Edition*. Prentice Hall PTR, 2 edition (april 24, 1999) edition, 1999. 2.3.1

[18] W. Live. http://www.live.com/. 1

[19] C. Macdonell and P. Lu. Pragmatics of virtual machines for high-performance computing: A quantitative study of basic overheads. 2.3.1

[20] Maui. http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php. 2.3.2

[21] M. messenger. http://services.nl.msn.com/messenger/. 1

[22] Microsoft. Ecma c-sharp and common language infrastructure standards. Internet. 2.3.1

[23] D. Mills. Network time protocol (version 3): Specification, implementation, and analysis. RFC 1305, July 1992. 3.1.1.2

[24] Moab. http://www.clusterresources.com/. 2.3.2

[25] J. Moy. RFC 1584: Multicast extensions to OSPF, Mar. 1994. 3.5.1.3

[26] myspace. http://www.myspace.com. 1

[27] T. network Library. http://twistedmatrix.com/trac/. 4.1.3.2

[28] parallels. http://www.parallels.com. 2.3.1

[29] Python. http://www.python.org. 4.1.3.2

[30] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, university of wisconsin - madison, October 2000. 2.3.2

[31] A. Silberschatz. *Operating System Concepts*, page 108. Wiley, New York, 2003. (document), 2.2.2

[32] S. Souders. *High Performance Web Sites*. O'reilly, 2007. ISBN:0596529309. (document)

[33] Sun. http://www.sun.com/datacenter/consolidation/index.jsp. 2.3.1

[34] A. S. Tanenbaum and M. van Steen. *Distributed Systems, Principles and Paradigms*. Alan Apt, 2002. 3.1.1.1, 3.1.1.2

[35] T. university of wisconsin. http://www.cs.wisc.edu/condor/. 2.3.2

[36] vmware. http://www.vmware.com. 2.6, 2.3.1

[37] W3C. The xmlhttprequest object, april 2008. 3

[38] D. Waitzman, C. Partridge, and S. E. Deering. RFC 1075: Distance vector multicast routing protocol, Nov. 1988. 3.5.1.3

[39] W. Zhang. Linux virtual server for scalable network services. (document), 2.2.2, A.5