UNIVERSITÀ CA' FOSCARI DI VENEZIA

DEPARTMENT OF ENVIRONMENTAL SCIENCES, INFORMATICS AND STATISTICS
MASTER'S DEGREE IN COMPUTER SCIENCE

MASTER THESIS

Replicated open source databases for web applications: architecture and performance analysis

Gianpaolo Silvestrini

SUPERVISOR
Orsini Renzo

11 March, 2014

Author's e-mail: g.silvestrini@dsi.unive.it

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: http://www.dsi.unive.it

To my dear friend Ezio Lucenti.

Abstract

Database system are currently used by thousands of web applications. Some of these applications serve critical services and have very high demands for high availability and fast real-time responses.
Open source database management systems, in the last years, have had some new features in terms of high availability and in terms of performance: although these are concepts largely covered by commercial database are new on the database open source such as PostgreSQL and MariaDB. In the past a cluster active-passive solution was the only way to have an high-availability architecture but with the new features other types of architectures are possible to obtain an high availability solutions.
The purpose of this thesis is to describe and compare the methods implemented on the main open-source database management systems to achieve high-availability.
I'll show the difference between the typologies of cluster and the downside of every solutions: since usually this type of technology are used from middle-size to large environment, i will demonstrate that with some disk-memory tuning on Linux based operating systems the performances can be significantly improved.
The acid test consists in a solution with two nodes and examines the throughput and the response time of a website (i.e. e-commerce or newspapers) and the heavy-write load in a TPC-C.

Contents

List of Figures

List of Tables

Introduction

I.1 Problem Description and Motivation

The demand of high performance and reliability on the database has always been important concept in the modern internet world.
In the past the common high availability was obtained on hardware level but it meant in most cases a downtime to reallocate the resource, i.e in a cluster solution where there is a master and a slave the downtime given is equal to reallocate time from the primary to the secondary host.
Modern Web application has to reach a near-zero downtime (or in the jargon 99.99% of availability), and this requirement has consequences on the hardware architecture and the software layer. Let's consider a internet-banking site in terms of availability: it is always available and the largest span of disservice is due to a scheduled downtime to deploy software's new releases. This is an example but there are many other cases of mission-critical web application, such as hospital or industrial ones.
Web application architecture is formed by three distinct actors: the web server, the application server and the database server. The best practice dictates that in front of each layer there must be a balancer distributing the jobs on servers of the same layer: this way let us have several identical servers doing simultaneously the same jobs. Regarding web server and application server layer, each server has an identical copy of "static" data, while data on a database layer are constantly modeled by the applications: that layer is the critical one of the web application architecture, because has to have synchronized servers.

In this sort of architecture is easy to understand that the chance of an availability's bottleneck is represented by the database layers and to avoid it we can implement different solutions, each one has different drawbacks. The easier solution is represented by a single database server on a virtual machine but ti means that we have a single point of failure at the application layer: in the event of a system crash, the database is unavailable. Moreover all maintenance operation may cause a downtime, and being a virtual system the resources given may be lower than a physical system.
Alternative solution of precedent options is have two or more database that works together, and in this case it can operate in synchronous or in asynchronous mode. Best solution in terms of availability is that the databases are synchronized with a balancer that divide the work load to these, but in this cases the performance can degraded. While the best solution in terms of performance has to have the database in asynchronous mode without balancer, but in this case is necessary a system downtime in case of failure, with

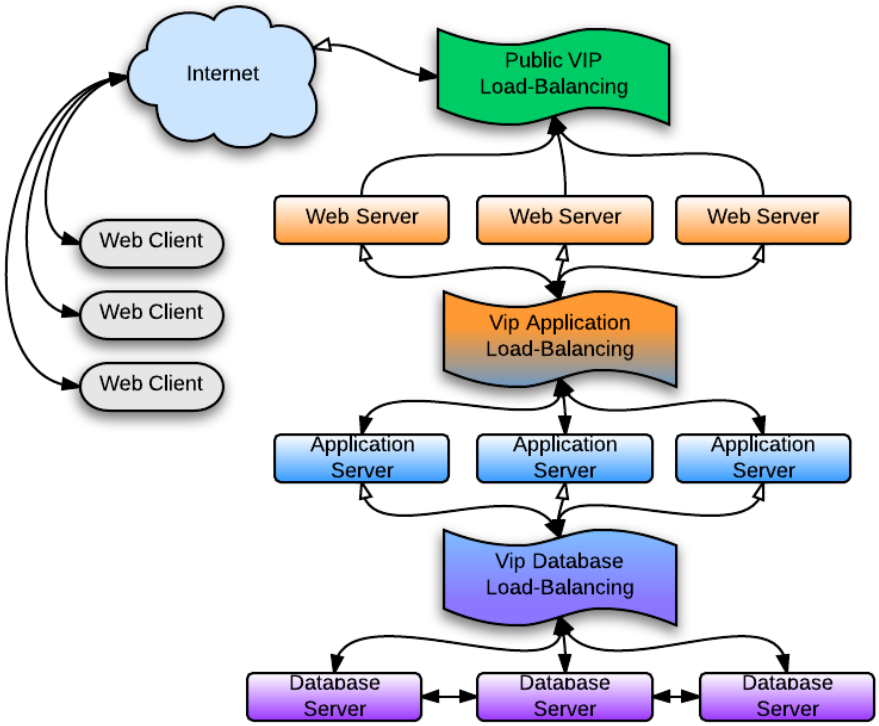


Figure 1: Traditional Web Architecture

a potential data loss.

The problem described below is a historical issue of the computer science. During the years the applications and the number of users on the web has increased and that has generated new problems and issues to guarantee the service level desired.
Nowadays there are several commercial database solutions that address the problem in a different way, but each and every one has high licensing fees and that's not an affordable option for every company.
Open-source world has two principal database management systems: *PostgresSQL* and *MySQL*. The last one became *Oracle*'s property after the acquisition [oraa] *Sun Microsystems* in 2010 and since then many new fork was born. *MariaDB* is the main DBMS born by *MySQL* and many companies (in private and public sector) use it for our scope [mara].

The best solutions to ensure high-availability of a database service are implemented in the commercial databases system for several years. This type of solutions consist to have different databases in synchronous mode with an virtual IP address (VIP) as single point of access.

In September of 2010 PostgreSQL has implemented, in version 9.1 [posc], a new feature to allow synchronous replication, and MariaDB team in 2013 has released MariaDB Galera Cluster 5.5.29 [marb] with the same feature. Before these releases the only way for high-availability was an asynchronous replication thereby the balancer feature was not an option.

The problem described is a real issue in many industrial realities. The historical moment we are living has showed an increasing number of companies that started using open source solutions. One of those is the IT Department of Veneto Region that since 8 years has begun to use the open source databases on its web architecture.
At first the databases were installed in a single instance, with no high availability system. After 5 years, as the system involved become more critical, an active-passive cluster was built, with an active node and an asynchronous second one available in case of failure. This solution has some problems such as the time to switch the database or the amount of time necessary to re-establish the initial condition after a fail-over. Obviously some others problems are at stake: the maintenance of the system is very problematic and it is mandatory to add others layers to manage the resource and the DBMS.

This work is born from the need to establish which solutions offer a real high-availability and which their benefits are. A customer likes to work in an environment that permits its maintenance without any collateral effects and usually his first request is a solution without any downtime.
The second request that a customer demands is that the database operates at its best. Ideally, his goal is an high-availability solution with two active nodes therefore the performances of the system are doubled: this is clearly a best case scenario, but an actual aim is to increase the total ranking of 80%.

I.2 Main Contributions

This thesis is an investigation of the new generation of open-source databases, and the type of synchronous replication system that these databases permit. As you may well notice this type of technology is available on the open-source product for a couple of years, and in literature there is not yet a performance comparison.
The contribution that we want to give to the community is a performance analysis of the synchronous replication on the open-source databases , in particular when these are utilized in a web architecture; to do so we will test several different types of benchmark in different situations, with an generic workload and with a web-based workload.

In order to achieve the thesis goal, is mandatory to know which open-source solution let us obtain the desired results. At the beginning, we have confronted the open source

database management systems and found which has the desired features. After that, the second step was understand how they work and how we must develop the architecture to obtain a high-available solution. During this phase, we have led a thorough study of every aspect of the examined DBMSes and a complete analysis of the problems discussed in this essay. Thereafter we have created the test environment that we have used to execute the tests. During this phase were created the three Linux machines used in this work: one of those has installed the load-balancing software and the benchmark, while the others two are the servers where the DBMSes reside. We did the setup of the DBMSes and the first exploratory tests to understand how the replication mode of each DBMSes really works.
Consequently, after we have fully understood how the DBMSes and each of their replication mode features work, we have began the balancing analysis phase: at this stage we have elaborated which are the load-balancing mainly used on the open-source world and which features they have. Thereafter during this step, we did the setup and the initial tests of the load-balancing software considered. We have also verified the host's performances where the load-balancing software has been installed: in fact, it is necessary to avoid bottleneck caused by the load-balancing machine and to achieve it we had to increase the resources of this host. The last part of the setup phase has been dedicated to install a resource statistics tool that is a basic element to comprehend the results obtained during the tests phase.
At the end of the setup phase, we have grasped how the the DBMSes and of the load-balancing software operate: it was possible to start with the real tests. We have done multiple several tests to measure how the performance of the different solutions changes. At this stage, we have tuned the hosts , in particular the I/O scheduler and the memory policy. When the test were completed, we have carried out the analysis of the results that showed which is the best test in terms of throughput and which is the bottleneck of any test.

The tests environment used is not simplistic: it replicates a real production environment where are involved a SAN storage system and a redundant gigabit Ethernet network. This type of environment makes possible to understand how the performance of database may vary if some operation system policies are modified, such as disk access and use of the swap. Eventually we will demonstrate in this thesis which is the best disk policy for any DBMSes when those use a SAN storage systems.

I.3 Outline

The remainder of this thesis is organized as follows. Chapter 1 provides an overview of what is a failure, error and fault, the concept of distributed system and the definition of database and several concept at the base of this object. In Chapter 2, is presented the meaning of high availability, and what is the mechanism to give it with hardware and

software solution. In Chapter 3 is presented the environment where the tests was been executed and which type of tests was done. In Chapter 4 is presented how the tests was performed and the result of these. Finally, Chapter 5 concludes this thesis.

1 Background

This chapter introduces high-availability and fault-tolerance terminology, A.C.I.D. proprieties and others key principles which are mandatory to elaborate this dissertation. The first part describes the Distributed Systems concepts and the latter lists several methods that can be used to improve the availability of database servers.

1.1 Failure, Error and Fault

As described in [Lap92], the functional safety of a complex system can be compromised by three types of incidents: failures, faults, and errors. When one or more of these occurs, the system can potentially be in state of accident.

Definition 1.1.1 (Failures) *A failure is the suspension of a functional units ability to accomplish a specified function. Since to complete a required function necessarily excludes certain behavior, and certain functions can be specified in terms of behavior to avoid, then the occurrence of a behavior to avoid is a failure [Com98].*

With previous definition, system can have three different states, because the concept of normal (safe) and abnormal (unsafe) can be removed. These are:

- **correct state**: there is no dangerous situation
- **incorrect safe states**: a failure was detected and the system is in a safe state
- **incorrect states**: this is a dangerous, uncontrolled situation: there are potential accessible accidents

Failures can be random or systematic. A random failure occurs unpredictably and is the result of damage affecting the hardware parts of the system. Generally , random failure can be quantified by its nature (wear, aging, etc.).
A systematic failure has a deterministic cause that can be eliminated by a reapplication

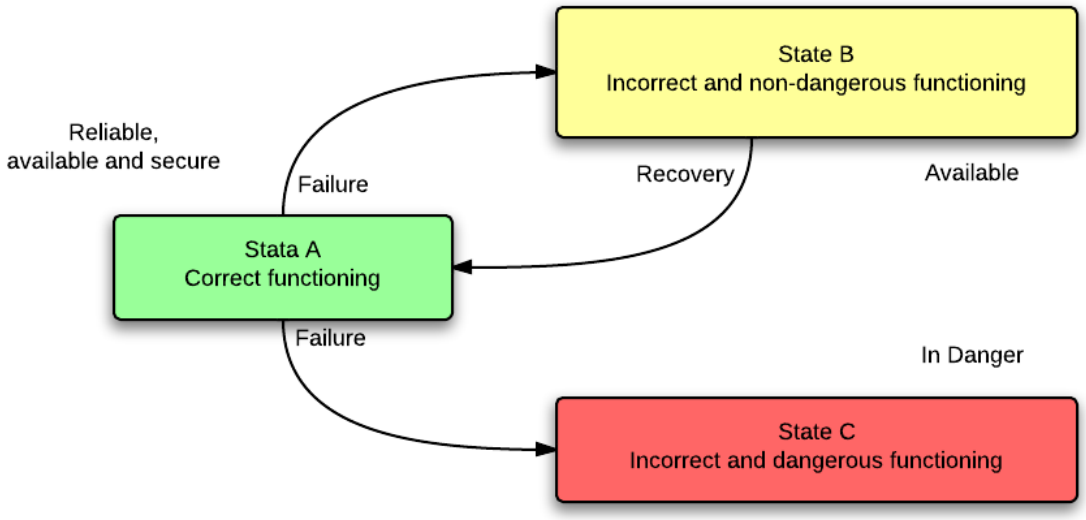


Figure 1.1: Evolution of system state

of the production process (design, manufacture, documentation) or by recovery procedures. Given its nature, a systematic failure is not quantifiable.

Despite all the precautions taken during the production of a component, it may be subject to design flaws, verification flaws, usage defects, operational maintenance defects, etc.

Definition 1.1.2 (Error) *An error is the consequence of an internal defect occurring during the implementation of the product (a variable or an erroneous program condition) [Com98].*

A failure can be caused by an error.

Definition 1.1.3 (Fault) *A fault is a non-conformity inserted in the product (for example an erroneous code)[Com98].*

A fault is the event that causes the error.

It should be noted that a safety state might be compromised by the appearance of obstacles such as faults, errors, and failures.

Example 1.1.1 *After three years of honorable service, an hard-drive has a faulty. Now if the computer try to read it, the HDD incurs in an error that cause a failure. Note that if the HDD is not read or is part of a raid solution, no error happens and there will be no failure.*

Every obstacle can trigger another obstacle, which eventually will derive in one or more errors: this (these) new error(s) may lead to the emergence of a new failure. The concept previously exposed can be easily brought to a higher level: as a matter of fact, if a system is in failure, it could cause a fault to another one which will be in its turn in a state of failure.

A fault is either soft or hard. In the first case, a *soft fault* apparently disappears after the failure and no repair is needed. In the second case, a *hard fault* does not disappear and the unit must be repaired to regain the normal efficiency of the system. The *soft fault* is also called *intermittent* or *transient*.

1.1.1 Hardware faults

One of the Major cause of failure is the fault of hardware components. Its failure rate varies over the lifetime. In the infancy period is high due to manufacturing defects. After a short time it becomes mature and the failure rate stabilizes on a low level. When the component has arrived at the end of the lifetime is worn-out and the rate increase again. Hardware reliability can be improved by removing the infancy and worn-out period. The

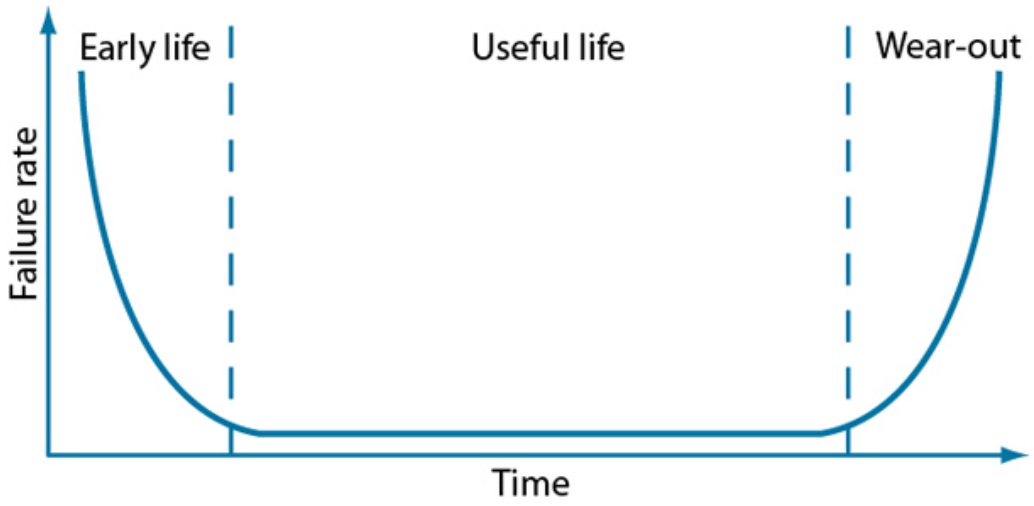


Figure 1.2: The Failure rate of the hardware components. It depends on the component life time and is similar to the bathtub curve.

first can be removed by running the hardware through a burn-in in the factory before deliver it to costumers, and the second reliability problem can be avoided replacing the hardware components when become close to decay. The operation environments has much influence on the hardware reliability. A rule of thumbs is that a temperature reduction of 10 degree can increase the lifetime with a factor two. Shocks, vibrations

and thermal changes cause mechanical stress on the hardware and reduces the lifetime and can directly cause a failure.

1.1.2 Software faults

An other cause of failure is software faults. It is not exposed to physical faults, only design faults. This means that once the problem is fixed, the fault is removed (in theory). The software faults are classify in two types: *Bohrbugs* and *Heisenbugs* [Gra85]. It can be considered as an analogy to hard and soft faults. The characteristics are:

- Bohrbugs: the fault manifests each time that the code is executed; are easy to single out and correct;
- Heisenbugs: some occasions tend to cause an error; are non deterministic;

”The trend of improving reliability of hardware has not been followed up by software technology and a larger fraction of failure is caused by software” [Gra90]

1.2 Distributed Systems

A system composed of independent physical *nodes* connected through a network is called *distributed system*. A node is a computer that include a software and hardware (not even special) components. Communication between nodes can be achieved by sending messages over the network. Users of such system don’t have knowledge of the node distribution because it is *transparent*. Message delivery can be *reliable* or *unreliable*. In the first case the communication channel has three main characteristics:

- It doesn’t lose messages. It means that the network is ”secure” and any message sent form one node is delivered to another.
- It doesn’t change messages. The message delivered is equal to sent sent one.
- It doesn’t copy messages. Any messages is delivery one time.

To save resource and increase the performance, the messages can be grouped together to save processing and communications overhead. This features is called **piggybacking** [CDKB11] and its purpose is increase the systems throughput at cost to delivery some messages until there are more going to the same destination.

1.2.1 Distributed Applications

In a distributed system environments the program executed on virtual node are called **distributed application**. The virtual node is a computer abstraction that spread out across the physical nodes (computers). For any physical nodes there are many virtual nodes that execute several distributed applications. Each virtual node offers a service to the others, and a user of the server is called **client**. However, the concept of a server is not static, because any server can be a client of on other server and so on. Obviously a single node can be both a server and a client, but it can be for two different separate requests.

1.3 Elastic Scalability

A Software system is said to be *scalable* if it is able to handle increasing load simply by using more computing resource. A system can be *scaled up* by adding more computing resources, such as memory, CPU or disk space. In a cluster environments we said that is *scale-out* if it allows to handle even larger workload by adding more physical machines. If the system can respond to changes in load by growing and shrinking their processing capacity on the fly, then it is an *elastically* scalable system.

1.4 Database definition

1.4.1 Transactions

A transaction is a unit of interaction with a DBMS and consists of any number or *read* and *write* operations and finishes with either *commit* or *abort*. Formally:

Definition 1.4.1 *Let $D = \{x_1, x_2, \ldots, x_n\}$ a data items stored in a database, $r(x_k)$ and $w(x_k)$ a read and write operation on data item $x_k : x_k \in D$ respectively, and let C a commit operation and A a abort operation. A transition T_i is a partial order wirh ordering relation $<_i$ where:*

1. $T_i \subseteq \{r_i(x_k), w_i(x_k) | x \in D\} \bigcup \{a_i, c_i\}$;
2. $a_i \in T_i \iff c_i \notin T_i$
3. *Let a_i or c_i, whichever is in T_i, for all other operations $o' \in T_i : o' <_i o$*
4. *if $r_i(x_k), w_i(x_k) \in T_i$ then either $r_i(x_k) < w_i(x_k)$ or $w_i(x_k) <_i r_i(x_k)$*

Implicit assumption of above model is that a transaction writes a particular data item only once time, and this is the reason why in the property four a pair of write operations is not considered.

History, conflict and A.C.I.D properties

To model concurrent execution of transactions *history* structure can be used. It indicates the relative order in which the operations of transactions have been executed. *History* is defined as a partial order if the operation might be executed concurrently.

Definition 1.4.2 (transaction history) *Let $T = \{T_1, T_2, \ldots, T_n\}$ a set of transactions, a complete history H over T is a partial over with ordering relation $<_H$ where:*

- $H = \bigcup_{i=1}^{n} T_i$
- $\bigcup_{i=1}^{n} <_i \subseteq <_H$
- $\forall$ *two conflicting operation* $p, q \in H$ *then* $p <_H q$ *or* $q <_H p$

By previous definition, a *conflict*[Pap86] between two transactions is defined as follow:

Definition 1.4.3 (Transaction conflict) *Let two different operations belonging to different concurrently executions transactions, read or write the same data item and not both are read operations, the corresponding transactions conflict.*

A first formal discussion of database transaction properties can be found in [Gra81]. From then a the formal transaction definition has led to give four key properties of a transaction: atomicity, consistency, isolation, and durability. The acronym ACID refers to:

- **Atomicity**: All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are. For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.
- **Consistency**: Data is in a consistent state when a transaction starts and when it ends. For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.
- **Isolation**: The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized. For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

- **Durability**: After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure. For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

In the distributed environment the transaction can be distributed: its characteristic is the execution of one or more operations individually or as a group and updates data on two or more distinct nodes of a distributed database. It is easily understandable that distributed transaction must provide ACID proprieties among multiple participating databases, which are dispersed among different physical locations; particularly, isolation property poses a special challenge for multi database transactions because the requirement of serial transactions execution is more difficult in a distributed environment.

1.4.2 Isolation Levels

In each database system, and especially in the distributed database environment, a distinctive interest resides in the *isolation property*. The Isolation guarantees that if a conflict between transactions is possible then the transactions are isolated from each other. There are different type of isolation level, and the ANSI SQL standard has specified four main level of isolation (ANSI 1992):

- serializable
- repeatable read
- read committed
- read uncommitted

The highest level of isolation is the *serializable* level. This level emulates serial transaction execution as if transactions had been executed one after another, serially, rather than concurrently therefore any overlapping is eliminated.
Repeatable read level provides that exist read and write locks acquired when the data has been selected, and these locks are released at the end of the transaction. This level introduce a problem, so phantom reads phenomenon can occur.
In *Read committed* isolation level the write lock has release at the transition end, while the read lock are released when the single operation is terminated. As for the previous level any lock are acquired when the data are selected.
Read uncommitted level is the lowest isolation level. It is possible have dirty read, so one transaction may see not-yet-committed changes made by other transactions.

Lower isolation levels are less restrictive but they can introduce inconsistencies during transaction executions. Less restrictive means also more performance. ANSI SQL

defines three different *read phenomena* that the transactions can present when T_i read data that T_j might have changed. These are:

- *Dirty read*: Assume a transaction T_i, modifies a data item, and another transaction T_j read the same data item before that T_i ends (commits or abort). if T_i performs an abort, T_j has read a data item that never really existed because it was never committed.

- *Non-repeatable reads*: Assume a transaction T_i reads a data item and another transaction T_j modifies or deletes that data item and commits. If T_i then attempts to reread the data item, it will receives a modified value or discovers that the data item has been deleted.

- *Phantom reads*: Assume a transaction T_i reads a set of data items satisfying some *search condition*. Transaction T_j then creates data items that satisfy T_i search conditions and commits. If T_i then repeats its read with the same *search condition* it gets a set of data items different from the first read.

The ANSI isolation levels are defined in terms of the above phenomena, and according to the ones they are disallowed to experience:

- *Read committed* prevents *dirty reads*

- *Repeatable read* prevents *dirty read* and *Non-repeatable reads*

- *Serializable* prevents *dirty read*, *Non-repeatable reads* and *Phantom reads*

Another recent and more utilized level added to the three common level defined by ANSI, is *snapshot level* [BBG$^+$95]. This new level avoids the phenomena defined in ANSI but exhibits inconsistent behaviour in some situations because it can produce other type of anomalies, as for example *write skew* [BBG$^+$95] and *read skew* [FLO$^+$05]. That consists in:

- *Write skew*: Assume two different data item z and y are related by a constraint that $z+y>0$ and the initial values of the two data items satisfy the constraint. In additional assume that the following order of operations on the transaction T_i and T_j is executed:

  $$r_i(z), r_i(y), r_j(x), r_j(y), w_i(x), w_j(y)$$

  It is possible that the transactions modify the data items in a way that the constraint is violated, such as T_i set x to 100 but T_j sets y to -150.

- *Read skew*: Assume two different data item z and y are related by a constraint that $z+y>0$ and the initial values of the two data items satisfy the constraint; additionally let's assume that the following order of operation on the transaction T_i and T_j is executed:

  $$r_i(z), w_j(z), w_j(y), c_j, r_i(y)$$

  It is possible that the reading of the sum $x+y$ by transaction T_i returns a result that violates the constraint.

1.4.3 Concurrency control, scheduler and correctness criteria

Any DBMS has a **Concurrency control mechanisms** to ensure that transactions are executed concurrently without violating the data integrity of a database: the aim of this component is to provide different degrees of isolation to transaction execution.
Other important component of DBMS is the **scheduler** that manage the overlapping executions of transactions. It receives operations from users and makes sure that they are executed in a correct way, in according to the specified isolation level.
This two component based our functionality on *servizability theory*, that has been developed to provide criteria to decide whether a history is serializable.
On the bases of this theory there's the concept of *equivalence*. It was introduced in order to provide syntactical rule to transform one history to another. With this theory it becomes possible to determine if two history have the same effect and if is serializable.
Concurrency control mechanisms has two main approaches to state the equivalence of histories and assesses if a history is serializable, and those are: *conflict serviceability* and *view serviceability*.

Definition 1.4.4 (conflict serializable) *Two operations are considered in conflict if they both access the same data item and at least one of them is a write operation.*

The rule of the concurrency control establishes the end result of any transactions, because it decides the order of execution of two conflicts operations. To ascertain which order impose to the conflicting operations it needs to know the dependencies of precedent operation.

Conflict Serviceability

This approach is usually applied to concurrency control of a single-version DBMSes. To decide which order impose to the operations, conflict serviceability use *serialization graph* structure. It captures these dependencies for a history H, and create a directed graph denoted as $SG(H)$ where the node are the transaction in the committed projection of the history, and the edge exist only if two transaction have at least one operation in conflict. Main features of the serialization graphs are:

1. Two histories are conflict-equivalent if their serialization graphs are identical.
2. A history is serializable if the serialization graph has no cycles.
3. Two histories H and Y are view-equivalent if:
   - they are over the same set of transactions and have the same operations.
   - the have the same *reads-from* relations, ie for any two committed transactions T_j and T_i and for any data item x, if T_i reads x from T_j in H then T_i reads x from T_j in Y
   - there have the same *final writes*, ie for each data item x, if w_i is the final write of x in H the it holds for Y too.

View-serializable and (S)P2L

View-servizability approach is used for multi-version DBMSs.

Definition 1.4.5 *A history H is view-serializable if for any prefix H' of H, the committed projection $C(H')$ is view equivalent to some serial history.*

Locking-based protocols are used to implement the isolation levels required. The common standard protocol used is *two-phase locking (2PL)*. It consist of two phases:

1. first phase acquiring locks without releasing any.
2. in the second phase locks are released without acquiring any others, and it does not require the ND of transaction.

The most safe protocol is *strict 2-Phases locking*(S2PL). It has been traditionally used to implement the serializable isolation level, though the ANSI SQL standard does not mandate its use. It avoids the phenomena described above. This protocol does the lock in read and write operation: a *shared lock* is created to read the data item and an *exclusive lock* when a write operation exists. The first allows only the access of concurrent reading, while the latter prevents both reading and writing of the same data item by other transactions. All locks are released at the end of the transaction, following the commit or abort operations.

Snapshot Isolation

Recent DBMSes implement *Snapshot* isolation to give a safe protocol like S2PL. A transaction executed in snapshot isolation operates on a snapshot of committed data that is taken upon at the begin of transaction. It guarantees that all reads operations is a consistent snapshot of the database; while the writing performed during the transactions

will be seen by subsequent reads within that same transaction.
A transaction can be aborted only due to *write-write* conflicts if operations try to modify data item(s) that has been updated by concurrent transactions. For example, Let a *begin* operation b and a commit operation c, and $b_i, c_i \in T_i$ and $c_j \in T_j$. This transaction are concurrent if the following holds:

$$b_i < c_j < c_i$$

As we can see there are conflicts in write operation, so this protocol is absence of conflicts between readers. It improves performance and make it more appealing than the other traditional serializable isolation level. In most DBMSes the Snapshot isolation is implemented using S2PL by to acquiring exclusive locks or writing data items. In commit phase the concurrency control mechanisms check *write-write* conflict using the following rules [FLO$^+$05].
Here under, it is reported the three liveness degree that provide some requirements for transaction commitments. These degrees are classified as greater as degree are stricter, and degree i embodies all other degree smaller that i:

- **First-commiter-wins**: if the lock is available T_i performs a version check against the executions of the concurrent transactions. Two outcomes are likely: if a concurrent transaction has modified the same data item and it has been already committed, T_i has to abort; otherwise it performs the operation.

- **First-updater-wins**: if the lock is unavailable, because another transaction T_j has an exclusive access, T_i is blocked. if T_j commits, the version check result in aborting T_i (First-commiter-wins). On the contrary, if T_j aborts, the resulting version check grants the lock to T_i so that it can subsequently proceed.

Version check that used the *First-commiter-wins* and *First-updater-wins* is called *version-creation-time* conflict check, because the key is that the version checks are performed at the same time that the transactions attempts to create a version.

An alternate possibility is *commit-time* conflict check. It doesn't use the S2PL but transaction execute on the particular snapshot in the *private universe*[FLO$^+$05]. This way the transaction acquires the lock, performs version check with the *First-commiter-wins* and transfers the version from the private universe to the database only at the end of the transaction. This approach brings unnecessary delay because it postpones the validation of the updates until the end of transaction.

1.4.4 Liveness

Replications protocols have more characteristics one of these is liveness. It can have a blocking or not-blocking behaviour. The latter ensure that every transaction eventually

terminates in commit or abort state. This propriety is not mandatory in fault-tolerant databases, but it is insufficient as it does not give any information on the *commitment* of a transaction [PG]. The simply protocol that wants to achieve the non-blocking property can create and aborting a sample transaction; if the system responds it can be considered live. In the following, it is reported the three liveness degree that provide some requirements for transaction commitments. This degree are classified as greater degree are stricter, and degree *i* embodies all other degree smaller that *i*

1. *Liveness 3* (the highest degree) ensures that every transactions commits. It is important to point out that this is something rather difficult to achieve.
2. *Liveness 2* ensures that read-only transaction are never aborted.
3. *Liveness 1* does not require anything about transaction commitment, though it ensures that every transaction eventually terminate (i.e. commits or aborts). This is behaviour of standard databases systems with non-blocking protocols.

Figure 1.3: Degree of liveness.

2

High Availability

The users should not only be satisfied with the specifications of the application that want to use, but should also know some detail about how it fails, how often fails, how much is downtime and the time recovery at normal state, etc. All this information is covered by the application's *dependability*.

Definition 2.0.6 (Dependability) *Computer system dependability is the quality of the delivered service such that reliance can justifiably be placed on this service. [Lap92]*

It is usual to assert that the dependability of a system should suffice for the dependence being placed on that system. Also the concept of dependence leads on to the one of trust, which can conveniently be defined as accepted dependence. In general dependability is the collective term used to delivery the quality of all aspects about how much the user (or others system) can trust (or depend on) a system. In fact there are many aspects which might be of interested not just a singular measure. The main attributes or measures for dependability are the following.

Reliability

Definition 2.0.7 (Reliability) *A measure of the continuous service accomplishments (or, equivalently, of the time of failure) from a reference initial instant. [Lap92]*

It is the probability that the system keeps operating correctly without failures for a defined period of time. The widely used definition is *Mean Time to Failure* **MTTF**. This is the average time from the system is set into service until it fails. An other possible measure is the probability of continuously service accomplishments over a given time interval.

Maintainability

Definition 2.0.8 (Maintainability) *A measure of the continuous service interruption, or equivalently, of the time to restoration. [Lap92]*

The measure for it is typically *Mean Time To Repair* **MTTR**. Considering maintainability on computer environment, it often includes ensuring that sufficient resources are available to the system, for example disk space or quite simply to apply software update. However, if we consider maintainability for the database, others characteristics are required. The backup is the major maintainability characteristic that the database needs. It should be atomic because is reflect the state of the database at a single point in time. Preferably the backup shouldn't affect the normal operations of the database system e.g. answering queries and performing updates.

Availability

Definition 2.0.9 (Availability) *A measure of the service accomplishment with respect to the alternation of accomplishments and interruption. [Lap92]*

With the reliability and maintainability definition, is possible derive the measure of Availability A as:

$$A = \frac{MTTF}{MTTF + MTTR}$$

It is a redundant measure because is computed form the reliability and maintainability measures but represents a single measure incorporating the essential information from the other two. Likewise, we can define *unavailability* U as the time that the system does not provide its specified service:

$$U = 1 - A = \frac{MTTR}{MTTF + MTTR}$$

The availability has been classified into the number of nines in the percentage of time it is available [GR93].

The definitions reported above does not consider the usage intensity of the system. As a matter of fact it might vary over time e.g. the system can have a higher request intensity during day than during night. A service disruption during a busy period would have worse effect that during hours with a low load. To satisfy for this deficiency, it should therefore be useful to have this alternative definitions that reflect usage frequency:

Definition 2.0.10 *Availability is the fraction of request that a system performs correctly and within specified time constraints.*

The availability is classified into seven different classes (view table 2.1). If A is availability its class C is expressed as:

$$C = log_{10} \frac{1}{1 - A}$$

Table 2.1: Classification of systems based on availability

System type	Unavailability (percent)	Availability (min/year)	Availability class (percent)
Unmanaged	50,000	90	1
Managed	5,000	99	2
Well-managed	500	99.9	3
Fault-tolerant	50	99.99	4
High-availability	5	99.999	5
Very-high-availability	0.5	99.9999	6
Ultra-availability	0.05	99.99999	7

For general purpose typically class 2 are request, while commercially availability fault-tolerant computer are class 4 and telecom switch are class 6.

Anyway, availability definitions is problematic when applied to query-oriented computer systems such as database which continuous availability is not easily defined, since availability of the systems is only measurable when a query is performed.

Safety

The last measure of dependability is not less important that the others. Safety is defined as:

Definition 2.0.11 (Safety) *A measure of continuous safeness, or equivalently, a measure of the time to catastrophic failure.[Lap92]*

Catastrophic failures are events that cannot be justified with the advantages that the system provides, but are independent events to the system. The quantitative measure for safety is *Mean Time To Catastrophic Failure* **MTTCF**. A system which fails gracefully avoiding catastrophic failures is called *fail safe* or *fail soft*.

2.0.5 Failure Classification

How described on chapter one, the system are prone to breakdowns. These failures can be classified into four different type of failures [Cri91]:

- **Omission failures**: The system (or server) is in hang status, so doesn't respond to any input.
- **Timing failures**: The system (or server) responds correctly but the respond arrives too early or too late. If it arrive too late, there is a *performance failure*.

- **Response failures**: The system (or server) responds incorrectly. There are two type of response failures: if the output is incorrect there's a *value* failure, while if there's an internal state transition incorrect is a *state transition* failure

- **Crash failures**: The system (or server) does not respond to input until it restarts. Is typical when omission failure happen. This failure has different branch; An *amnesia* crash has occurred if it restarts in the initial state. If after the restart the system has keep some state is a *parial-amnesia* crash, while is all state is retained before the crash is pause crash. Instead if server is never restarted after a crash a *halting* crash happen.

2.1 High system availability

The problem of achieving high system availability can be addressed in several different ways. The first technique that springs to mind is fault-tolerance, but it might not be sufficient by itself. There are others possibility to achieve the desired availability that can be faster and/or less expensive. Fault software or hardware are only one possibility for service interruption but also maintenance operations can cause downtime. In this section are explained the technique to give a high availability.

2.1.1 Fault-tolerance

As described in previous sections, fault is a reality and it is possible for the computer system to handle them without external interventions. To do this is necessary masking a fault so that it not cause a failure and the system (service) does not become unavailable. In general fault-tolerance is not a goal in itself, but rather a mean to achieve better dependability with respect to both availability and reliability.

The main keyword to fault-tolerance is *Redundancy*, either in time or space. *Space redundancy* is the number of modules used to process a task and when the same module does redundant work on a task by checking the result it is called *time redundancy*.

The redundancy can be used to the system to detect, mask and recover from faults. If there are two or more modules that have the same functionality, is easy to find if one (or more) is broke through the comparison of their results. If the result differs, at least one of the modules has failed and the failure can be detected. In case of more of two modules it is possible to find which is broken by taking a majority vote. When an error has been detected and eventually masked, the failed module has to recover into a consistent state.

Even though the hardware components are fault-tolerant, it cannot handle all kinds of errors. The single modules is only prepared to handle a known set of error and if an error is outside this set, the system will fail.

2.1.2 Fault-avoidance

It is better to avoid faults at all instead of having to handle them. After they have occurred it is difficult to correct and recover the system to the correct state. Avoid the fault can be do with simple attentions. These can be grouped as following:

- **High reliability**: Quality assurance, highly skilled designers and programmers must be keywords to keep in mind to have an high reliability and avoid the fault. Any component of the system should be designed with a high tolerance and resilience with respect to the environments.
- **Simplicity**: A simpler design contain less fault the complex ones. It is valid for hardware and software component (e.g. program with a less number of lines of code will reduce the number of bugs)
- **Automatic maintenance**: Plan automatic maintenance procedure can remove many human faults and help the recovery system.
- **Friendly user interfaces**: The human is one of major cause of fault, so implementing easy user interface reduce the number of human mistakes.
- **Environment Independence**: Each component interacts with others, but with less interactions there will be less hypothetical failures.
- **Security**: Malicious attacks can cause a failure and reduce the system availability.
- **Resource control**: Any component is designed to give a specific to handle a given load. Overloading can cause some problem e.g the system reject the new request and therefore threatens the availability. The system should handle overflows gracefully by rejecting requests, either by priority or at random. Alternatively, the system can be dimensioned with a sufficient over-capacity to handle the peeks.

2.2 HA though hardware

Hardware reliability is the main part to implement a highly available system. Nowadays the system that are entry, middle or enterprise level have the possibility to give each component redundant, and in most case any of this component can be replace with a live system. Individual modules, in turn, may have also been designed with internal redundancy. In this case the high availability is given by hardware redundancy and by software layer; Disk mirroring, redundant network connections is an example. Furthermore, over the years reliability of individual hardware modules has improved significantly due to improved designs and technological advancements.

In a *cluster* system the hardware redundancy can be formed by combining multiple physical machine or *nodes*. The cluster can be configured in one of the following mode:

- **Active/cold** configuration where one of the server is active and the second is a *cold* backup. In this case the scope is only high availability.
- **Active/passive** configuration where one of the server is active while the second is running the same workload (with some lag) and thus is a *warm* standby.
- **Active/active** configuration where both the server are serving workload actively. This solution give high availability and load balancing.

In general the more high availability and less inefficiency you want, the more expensive the systems are. Cluster solution can be geographically distributed in the cases that once server is active and others no. This provide better isolation from faults that may affect one site but not both sites at the same time. This solution unfortunately has some drawbacks. First, it requires in any sites, redundant hardware such as disks for mirroring and replications, Storage Area Networks (SAN), Network Area Storage (NAS), or multi-path storage devices for shared storage, and bonded network interfaces for added reliability. Second, we cannot just take any application and run it on a cluster and expect it to be highly available. Applications need to be made *cluster-aware* to exploit high availability features. This is the case of database system that are different in design and functionality for standalone and cluster solution such for example concurrency control, recovery and cache management. This add to the cost and complexity of building and deploying *cluster-aware* applications. Moreover there is the configuration and operation cost of a cluster that is much higher that a single system, and not least the licensing cost that in more cases is prohibitive for small or medium sized businesses.

2.3 HA for database systems

The previous section has mentioned that in case of high availability for database system, the hardware solution is not sufficient because it faces only that type of faults. High availability for database systems is given by hardware and software solutions. Although the type of cluster are three, the two common approaches for providing high availability for database system are two, because the active/cold solution is given by hardware redundancy (multi-site) and by application solution given by system operation.

2.3.1 Parallel database system

Parallel database systems typically running on clusters of node that are in the same site, and are a popular choose for enterprise solution with closed source software. The

cluster typology used in this type of system is active/active. A parallel (or clustered) database system executes database operations (such queries) in parallel across a collection of nodes. Each node runs a copy of the database management system (DBMS). Data can be partitioned among the nodes, and in this case each nodes is owned by a particular partition, or it is shared by all nodes. In all cases, any nodes have a copy of all data. Failure of individual nodes in the cluster can be tolerated and the data can remain accessible. Parallel database system is used to give high-availability, fault-tolerant an in addition to improved performance. Parallel database architectures in use today are mainly divided into *shared nothing* and *data sharing*.

Shared nothing (SN) architecture have the data partitioned among nodes and each nodes hosts have one or more partitions on its locally attached storage. Obviously if the cluster has two nodes, each has the same data of the other node. Access to these partitions is restricted to only the node that hosts the data and so only the owner of a data partition can cache it in memory. This type of implementation is simply because the data ownership is clearly defined and there is no need for access coordination. This type of solution have been a huge commercial success because they are easier to build and are highly scalable [Sto86]. An example of SN database systems include IBM DB2 [PCZ02], Microsoft SQL Server [Cor], MySQL Cluster [Orab], Postgres-XC [posb] and Mariadb with Percona or Galera cluster [Per] [Cod].

Data sharing (DS) architecture data are shared and accessed by each nodes. Data is usually hosted on shared storage as for example by NFS share or by using SAN. Since any node in the cluster can cache and access any piece of data in its memory, is needed the access coordination in order to synchronize access to common data for read and write operation. This operation a requires distributed locking, cache coherence and recovery protocols with add the complexity of a data sharing system. More complexity means have more overhead because access coordination must be ensure that the data are consistent between all member of the cluster. Main advance of DS is that do not require that the database is partitioned and thus have more flexibility in doing load balancing. An example of DS database systems is Oracle Real Application Cluster (RAC) [Orac] and mainframe IBM DB2 [IBMa].

2.3.2 Replication database systems

Replication database system is a technique where database is fully or partially replicated locally or at a remote site to improve data availability and performance [ÖV11]. Any database copy is referenced as primary (or master) replica if is the original copy of the data, while is called secondary replica if is the replicated copy. If the master replica becomes unavailable, the data still remains accessible through the secondary replica. The system can also have N-way replication where a primary replica is copied to N-1 other replicas. More nodes the cluster has, the greater will be the overhead. Replication can

be implemented into a single DBMS, or between multiple DBMS across more machines that can be in single-site or multi-site. Changes from the master node to the secondary nodes are periodically propagated. The type of propagation for keeping the secondary replica up-to-date with the primary replica can be *synchronous* or *asynchronous*.

Synchronous (streaming) replica is the reference method when the organization needs *zero data loss*. The systems thus becomes highly availability where in the event of a failure of the primary data source, another mirror system should take over immediately from the latest point of failure because it offers the advantage of keeping all the replicas strongly consistent. However, due to the high operational costs associated with acquiring locks, and excessive massage passing for two-phase commit, it is rarely used in practice. This method consist to have all DBMS with a exact copy of database. For this type of replication, the updating transaction must acquire exclusive access to all the replicas which might involve lock requests across remote sites. This type of replication is traditionally used over short distances, or when transmission delay between all nodes is very low.

Asynchronous replica offers a trade-off in terms of minimizing overhead of normal operation and data consistency. With asynchronous replication, it is possible for a transaction to get slightly different results when accessing different replicas because they are updated only periodically. Although asynchronous replication has the advantage of speed, there is an increased risk of data loss sing this method, because received data is not verified. In fact the master replica sent the update on the other node but doesn't wait to have a response (positive or negative). Asynchronous replication slows writes to the source volumes while the target volumes are update in the background. Typically, it is used primarily in disaster recovery scenarios where the recovery site is located far away and the application would experience severe performance degradation with synchronous replication.

Motivation

The basic reason why the replication (sync or async) is implemented, is to give high availability of the system and in some cases to improve the performance; This concepts are the basis of disaster recovery planning and are the business requirement of the any organization. The need to know what is being protected and why is essential for running the organization. A well-documented requirement in the form of *Business Continuity Plan* spells out these clearly taking in to account *Risk Assessment* and *Business Impact Analysis*.
Building a replication solution must meet that the primary objectives the *Business Continuity Plan*.
Every organization must spell out its *Recovery Point Objective* (RPO) and the *Recovery Time Objective* (RTO) to get the application running. The *Recovery Point Objective* (RPO) is the maximum acceptable level of data loss following an unplanned event, like

a disaster (natural or man-made), act of crime, or any other business or technical disruption that could cause such data loss. This represents the point in time, prior to such an event or incident, to which lost data can be recovered. with the *zero data loss* (ZDL) objective a business would like to have the up to data available in the event of failure due to say a disaster recovery, hardware failure or any other disaster. Of course you get the ZDL, a synchronous replication solution must be used.
The *Recovery Time Objective* (RTO) is a period of time within which business must be restored following an unplanned event or disaster. The RTO is a functions o the extent to which the interruption disrupts normal operations and the amount of revenue lost per unit of time as a result of the disaster. These factors in turn depend on the affected equipment and applications.

General architecture for replication

As said earlier, data replication is the process of generating, reproduction, and maintain a defined set of data in one or more location [Haa01]. The benefits of data replication include improved performance when centralize resources get overloaded, increased data availability, and capacity relief.
It terms of architecture, several model exist and the main characteristics that divide these in two groups are if the model are synchronous or asynchronous.
Following the main architecture for replication:

- **Master-master** replication model, have multiple sites with updating data privilege [Kea09]. Also referred as multi-primary or multi-directional replication, this kind of model requires detection and resolution of update conflict. *Update conflict* occurs when different sites try to commit competing or incompatible updates to each of their own replica; below a more accurate description.

- **Master-slave** replication model, has only one of the data locations privilege to read ad write data. All the rest of the replicas are in some cases read only, and in others cases not action can be performed. In this way, all updates to the data are committed through a central/primary location, and then are replicated to backup or ”slave” sites. This model is also called *primary-backup* replication or *single directional* replication. Obviously the problem of update collision disappears in this kind of systems.

Compared to master-slave replication, multi-master replication models can continue to update database when one of the master sites fails. The resource utilization rate in also higher in multi-master systems, as any replicated system can perform updates to databases. On the other hand the disadvantages of multi-master model are that conflict resolution may introduce extra communication latency in synchronous replication systems, while asynchronous replication systems we lose the advantage of data consistency.

Models of replication

The replication models are how to the replication is done on the systems. The types are different if the replication is synchronous or asynchronous. Today there are three different type:

- **Snapshot** replication is taking an entire set of data and replicating it to another database. This method is most powerful and easy to set up replication type. Anyway, this method is typical used periodically because the data size grows with every replication, becoming difficult to manage, time and resources consuming. Another typical use is when the replication model is installed, so the first replica nodes is a snapshot of the primary. This is in general an asynchronously replication model.

- **Log-based** replication model, propagates only changed data instead of the entire data set in database. Changes are detected by scanning log maintained by the database management systems (DBMS) and then sent to other replication locations. Typically this model operates asynchronously, however it has the advantage of low overhead against other change capture strategies since it only uses the existing logs in the DBMS. The DBMS change detection and the replication processes don't interfere withe the normal operation of the source database system, because these are background processes.

- **Trigger-base** capture has a model where any modification to data has been registered for replication set off triggers inside the database. Those triggers in turn either directly activate the replication process or push changed data into some kind of queue. This model can either operate asynchronously or synchronously. Since the processing of triggers happens at the same time as the processing of users transactions, performance of the source databases is affected directly.

Conflict and deadlock

When the replication system is synchronous, the update-everywhere replication has a high conflict rate and a high probability of deadlocks. In literature, some studies [GHOS96] showed that in some situations the probability of deadlocks is directly proportional to n^3, where n is the number of replicas. The observation is not surprising: as the number of replicas increases, the time to lock the resources will increase too and transaction execution times will deteriorate. Also, the longer transaction time caused additional communication overhead when the number of nodes increase. Evolution of deadlock is the *distributed deadlock*. Certain database replication solution are prone these more complicated deadlock. This happen if resource lock is acquired in different order on different replicas.

Example 2.3.1 (Distributed deadlock) *Suppose to have two concurrent transaction T_1 and T_2 witch are competing for resource A while executing in a replicated database system with two replicas R_x and R_y How to show in figure 2.1 the order of lock requests by T_1 and T_2 is different on the two replicas. In the figure it is possible to notice that the*

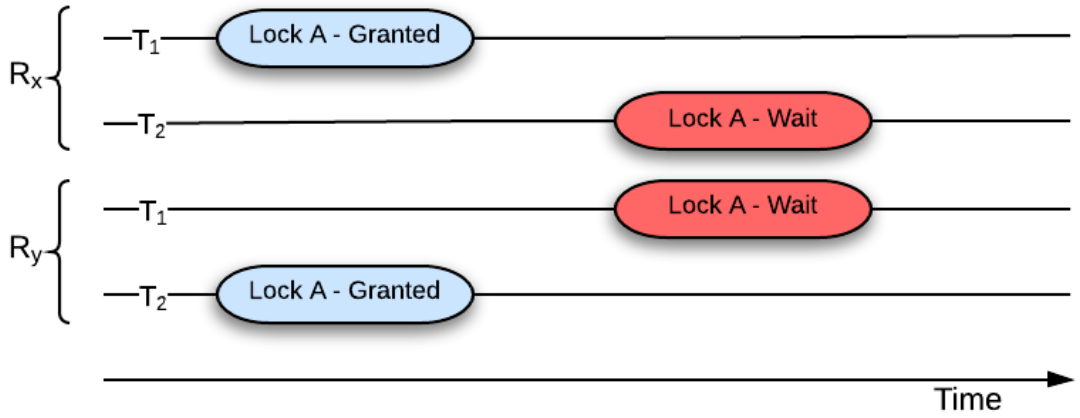


Figure 2.1: Example of distributed deadlock

transition T_2 is blocked waiting for T_1 on R_x and vice versa is true on R_y. As a result, in replication schemes where a transaction commits only after all the replicas are ready to do so, the transactions would be deadlocked without possibility to progress further.

To avoid this type of deadlock, it has been suggested that a *group communication systems* be used as a means of reducing conflicts as well as ensuring consistent data on multiple replicas [HT93]. The group communication systems are capable of ensuring that a messages multi-cast in group will be delivered in the same total order on all nodes of the systems.

3

Database Architecture Test

In this chapter we present the databases that are used to execute the test of high-availability and performance evaluation. We describe the background infrastructure in which we have conducted the tests and the type of load balancing used to compare the *master-master* and a variant of *master-slave* architecture for replication.

3.1 Infrastructure environment

Enterprise environments was chosen to execute all the comparisons between the different DBMS and replication architecture. In absence of dedicated physical server, we've utilized a virtual environment.
Our environment is composed of three Linux hosts: Inside the first two there are the DBMSes, in the third we have configured the "load-balancing" software and it's where the benchmark starts. To replicate an high fidelity production environment, all hosts are *Red Hat Enterprise Linux 6.4 64-bit* [Hat]. Virtual Hosts are inside three different nodes of a VMWare HA cluster [VMW] that are managed by a *VMWare Virtual Center*

3.1.1 Virtual environment

VMWare vSphere HA Cluster is the virtualization technology that has was adopted. The physical cluster is composed of three nodes where the virtual machines reside. Physical node are the *IBM HS22* with 72GB of RAM and two cpu Intel X5570 without physical hard drive, while the others characteristics are describe below:

Storage

The storage utilized by the physical nodes, as for the rest of environment, is an enterprise one: all storage are attached to the Storage Area Network (SAN), composed by two main parts.
The first one is the real storage: it is a *IBM DS8000* [IBMb]; the disks have a capacity of 450GB each and 15.000 RPM (Revolutions per minute). In the sake of a better

performance and reliability, *IBM DS8000* joins the disks into a RAID 10 array solution. Management of the array is transparent for any client host that is attached to the storage system.
The second part is the *IBM System Storage SAN Volume Controller* [IBMc]: it's a storage virtualization system with a single point of control for storage resources. This means that all resources of the real storage are managed and optimized by this system.

Each of the previous components, has an owner cache that permits to increase the throughput of most frequently resource requests. This type of gain is not possible with a traditional storage like a SAS disk.

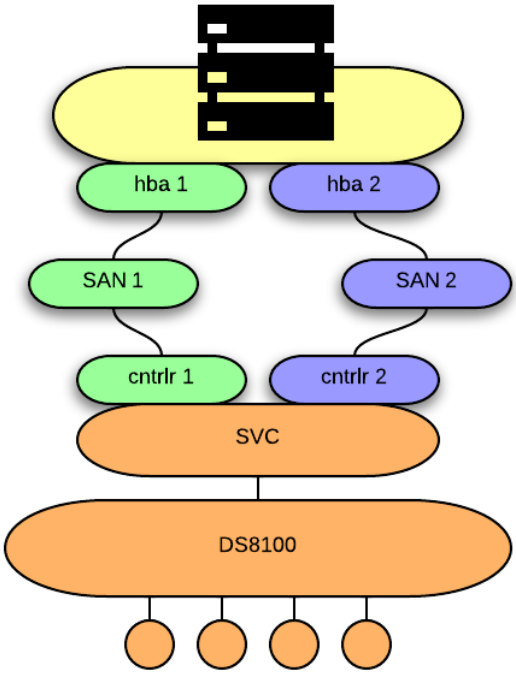


Figure 3.1: Active/Active Multipath Configuration

Each nodes of VMWare cluster is attached to the Storage Area Network. The SAN is composed of two redundant part called *Fabric*, that improve the high-availability and the performance of the storage. Redundancy means that we'll have for each device attached to the SAN, a minimum of one connection for *Fabric*; obviously this statement is valid also as for the real storage as for the storage virtualization.
In particular the physical nodes of the VMWare cluster have two fibre channel HBA (host bus adapter) with 4GB dual-port each. Any HBA port is attached in one different *Fabric*.

The scheduler access adopted in VMWare for storage attached by SAN is a *Round Robin* type. It means that automatically detects the loading on the I/O paths, and dynamically re-balances the load. This is possible because the configuration of VMWare support active/active path, i.e. all paths active with round-robin load balancing. So the final maximal throughput that each physical VMWare node has toward the storage is $(4+4)*2$ GB.

Network

Network layer has a significant relevance on the database system because all communications between the server and the client are transmitted trough the network.
The Virtual machine has two virtual gigabit Ethernet port on different network. Main network is public and utilized to communicated with the rest of world, such as client and others server, while the second network is a heartbeat and it' a private one. Physical host instead has three real gigabit Ethernet port. These are aggregated with the IEEE 802.3ad protocols [IEEa] to avoid a *single point of failure* and to improve the performance.

One feature on a virtual environment infrastructure as the one in VMWare is to man-

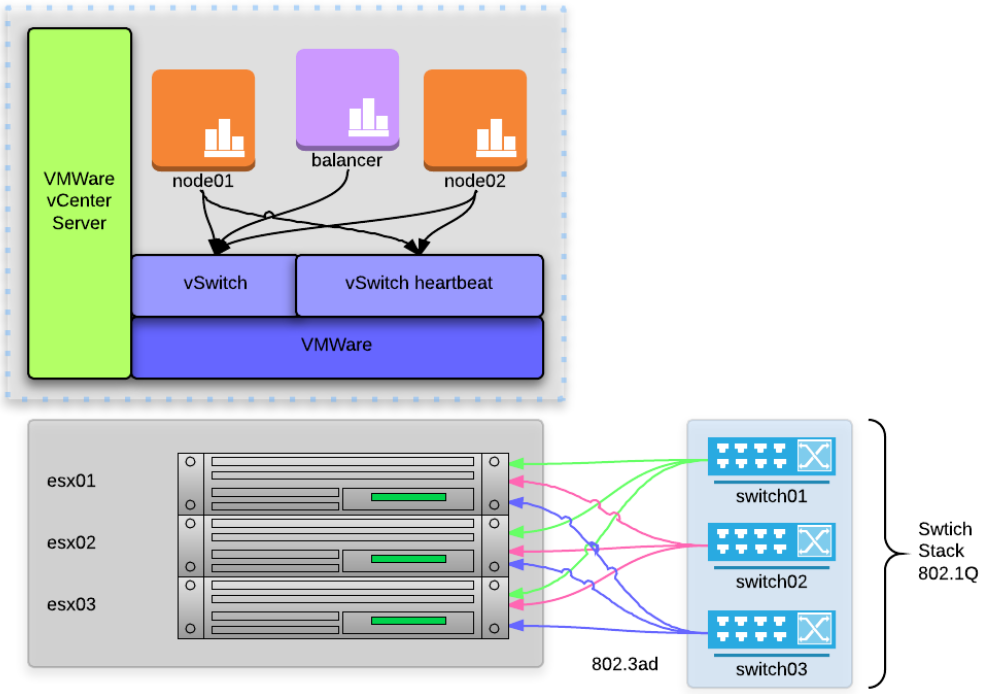


Figure 3.2: Pysical network configuration

age more different network with a restrict number of Ethernet port. It is possible with

the IEEE 802.1Q protocol [IEEb] to trunk more VLAN on the same physical cable(s). VMWare infrastructure can create on the physical NIC more *virtual switch* and those tag the VLAN present on the virtual machines. In this environment a physical host has 3 NIC and almost two virtual switch to ensure that the private and public VLAN is separated.

3.2 DBMS

Aim of this thesis is evaluate difference on open source DBMS it terms of performance and high availability. The DBMSes have been chosen about the world popularity and because an background goal is a future convergence of the application of the currently existing.
The actually application has as DBMS PostgresSQL 8.4 [posa] and MySQL 5.1 [mys], so these are the DBMS target.
In 2010, *Oracle Corporation* has purchased [oraa] *Sun Microsystems* and its open technology such as *Java*, *SPARC Enterprise* and *MYSQL*. From that acquisition, the principal developer of MySQL has forked and other open source DBMS: MariaDB [mara]. This new DBMS is used by many users, ie *Wikimedia fondation*, *Oracle* and others big.

3.2.1 MariaDB

The biggest strength of MariaDB is, of course, its compatibility with MySQL. It is a drop-in replacement for the same version of MySQL; that is, if an application works with MySQL 5.5, it should work with MariaDB 5.5 without any modification. Obvious precedent version also works without any modification.
The core of any database system is its data storage engine, and MariaDB offers several powerful engines to choose. The current default storage engine is InnoDB, and it's the storage engine used in this thesis. Key advantages of InnoDB is that is designed for the ACID properties: its transactions has commit, rollback and crash-recovery capabilities to protect user data. Whenever there's an update, insert or delete the lock is at row-level and not at table-level. This is more important when the database is distributed, because it's less probable that two transactions at the same time modify the same row than two transactions working at the same time on the same table.
Another characteristic of InnoDB storage engine is its own buffer pool for caching data and indexes in main memory. It stores its tables and indexes in a tablespace, which may consists of several files.

MariaDB can be a cluster solution if it is installed with *Galera wsrep* solution. This component is a GNU GPL [gpl] solution that provide the following features:

- No fail-over requirements - the nodes are all active and are all masters

- The application can read and write to or from any server
- Horizontal Scalability (scale out) for both reads and writes
- Automated online add node
- Easy node removal (scale in)
- No application change scale out
- Resilient to high latency networks
- No data lost

To utilize this feature it is necessary use an optional load-balancer whenever the application connects to any server of the cluster and starts a transaction. When the application commits, all the data changed within the transaction are moved to the other nodes of the cluster. The commit will complete only when all the nodes have received the data, which is applied locally at a later time. In this way the impact in terms of performance is very limited.

3.2.2 PostgreSQL

One of the two most advanced object-relational database management systems free in the world is PostgreSQL. It is developed by PostgreSQL Global Development Group under GNU GPL license [gpl].
PostgreSQL supports most of the SQL standard and offers many other features such as: complex queries, foreign keys, triggers, views, transactions integrity, multi-version concurrency control: it doesn't lack on any important feature that its competitors have, ie an high availability in active/active solution. In fact, PostgreSQL do not has a native asynchronous solution such as the log shipment or others low-level methods, and only in recent versions (from 9.1) it able to have a slave server in read-only state synchronized with the master server. This type of replication is the *master-slave* architecture with the *log-based* model replication.
Load-balancing is not a PostgreSQL feature, and the common software balancing is not sufficient because if is implemented a PostgreSQL streaming replication, is possible to commit the database only from the master node. Third-party solution has been developed to give this type of functionality, and the most used in the world is *PgPool-II* [pgpb].

3.3 Benchmark

After the execution of benchmarks of the DBMS, with an overview of a global throughput score, developer or/and database administrator can investigate where the system bottleneck is. As demonstrated in many studies, two are the major limitations of a system: the disk layer and the lack of memory. Nowadays the latter has less relevance than many years ago, because the cost of the RAM is decreased. The first problem can be bypassed with numerous and more efficient disks, like SSDs, but the problem, although to a lesser extent, remains. One of the objective of this thesis is to understand how the system changes if we change the storage layer and if there is an optimal solution for any DBMS.
Following we describe the types of I/O scheduler chosen on the test, and how to the operation system manage the memory swap.

3.3.1 I/O scheduler

As previously sated, the I/O scheduler has an important role in any computer system. Actually there is three major scheduler implemented on any Linux kernel: *noop*, *deadline*, *cfq*. These burn in different times and with different reason, and principles and differences are:

- **Noop**: This scheduler assumes that either the request order will be modified at some other layer of the operating system or that the underlying device is a true random access device. When the noop scheduler receives a request, it first checks to see if it can be merged with any other outstanding request in the queue. If it can be, it merges the two requests and moves onto the next request. If no suitable merge can be found in the queue, the new request is inserted at the end of an unordered first-in-first-out (FIFO) queue and the scheduler moves on to the next request.

- **DeadLine**: It is based on noop Scheduler, but add two important features. First, the unordered FIFO queue of the noop scheduler is replaced with a sorted queue in an attempt to minimize seek times, and second, it attempts to guarantee a start service time for requests by placing deadlines on each request to be serviced. To achieve this, the deadline scheduler operates on four queues; one deadline queue and one sector sorted queue for both reads and writes. When a request is to be serviced, the scheduler first checks the deadline queue to see if any requests have exceeded their deadline. If any have, they are serviced immediately. If not, the scheduler services the next request in the sector sorted queue, that is, the request physically closest to the last one which was serviced.

- **CFQ**: it places all synchronous requests which are submitted into one queue per process, then allocates time-slices for each of the queues to access the disk. The queues are then served in a round robin fashion until each queue's time-slice has expired. The asynchronous requests are batched together in fewer queues and are served separately.

To summarize, the noop scheduler is so called because it performs a bare minimum of operations on the I/O request queue before dispatching it to the underlying physical device. It cannot guarantee a specific service time for any given request. Since different applications may have different performance requirements, service time guarantees may be desirable. To address this shortcoming, the deadline scheduler was created. This scheduler generally provides increased performance from the noop scheduler due to the fact that the sorted queues attempt to minimize seek times. In addition, by guaranteeing a start service time, the deadline scheduler can provide service time guarantees.
Completely Fair Queuing scheduler instead, is the default scheduler in the Linux distributed and attempts to provide fair allocation of the available I/O bandwidth to all initiators of I/O requests.
Due the type of tests and the type of infrastructure, the tests were performed with the noop and cfq scheduler. The deadline is not been considered because it is suggested for real time application.

3.3.2 Swap Memory

Linux Kernel has the possibility to swap out run-time memory. This propriety permit at the system to have an aggressively use of swap user or vice-versa try to avoid swapping as much as possible.
Swappiness parameter can be set at run-time and can be set with values between 0 and 100 inclusive. Low value means the kernel will swap only to avoid an out of memory condition. At the opposite an high value means that the kernel will swap aggressively which may affect over all performance.
The default value is 60, and in this thesis we tested the behaviour with a default value and with 0 value.

3.3.3 Load Balancing

With the DBMSes described in the previous chapter, it is clear that to have a real high availability and a real parallel distribution of the load, is necessary to have a middleware layer between the client and the server. Unlike the commercial solution, where the DBMS has a layer that manage the load and the reliability of the system, the free solutions such as PostgreSQL or MariaDB haven't this layer. Fortunately this is not a problem because other free solutions exist to avoid this deficiency.

The main goal of load balancing is improving the performance by balancing the loads among computers. To do this is necessary that the load balancing know witch are the server live, and which is its load.
In literature the load balancing are divided into two main category policies: *static policies* [CK88] and *dynamic policies* [RR96].
Static policies base their decisions on statistical information about the system. They do not take into consideration the current state of the system. Dynamic policies base their decisions on the current state of the system. They are more complex than static policies.

The load Balancing used in this thesis have static policies. Their primary goal is to test the maximal throughput of the system. This type of software has its workload, and this is the reason for which a dedicate host exists, where the tests are started and the load balancing is installed.

The load balancing used are three, and was chosen because is the most reliable solution and the most used on the real production environment. Two of three are used to test MariaDB DBMS, while the last one is used to test PostgreSQL. This chose is forced because PostgreSQL DBMS is not able to have a *active-active* solution in *write-all*, and so we need to use a dedicated load balancing.

Galera Load Balancing

Galera Load Balancing (GLB) is a TCP load balancer. Its aim is to make a user-space TCP proxy which is as fast as possible. It is an open source project and it is licensed over the GNU GPL license [gpl].
This balancer has the following features:

- list of back-end servers is configurable in run-time.
- supports server ”draining”, i.e. does not allocate new connections to server, but does not kill existing ones, waiting for them to end gracefully.
- It is multi-threaded, so it can utilize multiple CPU cores. In fact even on a single core CPU using several threads can significantly improve performance when using poll()-based IO.
- Can monitor the health of destinations by polling using service-specific scripts and adjust routing table automatically. In Galera cluster it also can (optionally) discover newly added nodes and take them into use.

GLB supports five balancing ”policies”:

a) **least connected**: new connection will be directed to the server with least connections (corrected for server ”weight”). This policy is default.

b) **round-robin**: each new connection is routed to the next destination in the list in circular order.

c) **single**: all connections are routed to a single server with the highest weight available. All routing will stick to that server until it fails or a server with a strictly higher weight is introduced.

d) **random**: connections are distributed randomly between the servers.

e) **source tracking**: connections originating from the same address are directed to the same server.

Galera Load Balancing is the unique balancer that has a special feature that works only with the *Galera wsrep* solution. It is a destination discovery features, that allow, if destinations can supply information about other members of the cluster, to automatically populate watchdog destination list.

HA Proxy

HA Proxy is TCP/HTTP load balancer. It is the historical software balancer used in Linux like system. Unlike the previous balancer, it has the ability to improve the performance of web sites by spreading requests across multiple servers, because it can be a HTTP load balancer. This functionally is not used on this thesis, and it is used only with the TPC balancing.
Many vendor utilized HA Proxy for the overt performance. *HAProxy implements an event-driven, single-process model which enables support for very high number of simultaneous connections at very high speeds. Multi-process or multi-threaded models can rarely cope with thousands of connections because of memory limits, system scheduler limits, and lock contention everywhere. Event-driven models do not have these problems because implementing all the tasks in user-space allows a finer resource and time management. The down side is that those programs generally don't scale well on multi-processor systems. That's the reason why they must be optimized to get the most work done from every CPU cycle.* [hap]
This product has more feature that of those required to this study. It is used in this thesis for load balancing a TCP connection over the two DBMSes. Additionally feature such as a watchdog or cli management are used indirectly to executed the test.

PgPool-II

Pgpool-II is a middle-ware that works between PostgreSQL servers and a PostgreSQL database client. It is licensed under BSD license [bsd]. This software doesn't have only the load-balancing functionality, but implement more feature that PostgreSQL haven't:

- **Connection Pooling**: To reduce the effort that any new connections has, *Pgpool-II* saves connections to the PostgreSQL servers, and reuse them whenever a new connection with the same properties. It reduces connection overhead, and improves system's overall throughput.

- **Replication**: Previous PostgreSQL version haven't the replication capacity between multi-server. This features can manage multiple PostgreSQL servers. Using the replication function enables creating a real-time backup on 2 or more physical disks, so that the service can continue without stopping servers in case of a disk failure.

- **Load Balancing**: If is present a native streaming replication, executing a read-only query on any server will return the same result. *Pgpool-II* takes an advantage of the replication feature to reduce the load on each PostgreSQL server by distributing SELECT queries among multiple servers, improving system's overall throughput. At best, performance improves proportionally to the number of PostgreSQL servers. Load balance works best in a situation where there are a lot of users executing many queries at the same time.

- **Limiting Exceeding Connections**: There is a limit on the maximum number of concurrent connections with PostgreSQL, and connections are rejected after this many connections. Setting the maximum number of connections, however, increases resource consumption and affect system performance. *Pgpool-II* also has a limit on the maximum number of connections, but extra connections will be queued instead of returning an error immediately.

- **Parallel Query**: Using the parallel query function, data can be divided among the multiple servers, so that a query can be executed on all the servers concurrently to reduce the overall execution time. Parallel query works the best when searching large-scale data.

3.3.4 Oltpbenchmark

When a new DBMS is installed or when an DBMS upgrade is done, bench-marking is the first operation that any database administrator want do. This is often overlooked but the role and the benefits that it can bring is very important to prevent a several issue performance on production environment.
Into database world exist many different types of benchmark, that can be execute carry-on or by a specifics tool. When I began this thesis, the first problem that i have addressed was which tools use to execute the test, and which test i would have execute. This problem was resolved with the *Oltpbenchmark* tool, that is a *an extensible batteries included DBMS bench-marking tested that tailored for on-line transaction processing*

(OLTP) and Web-oriented workloads [oltb]. This tool provides an aid to compare in easy way the result, and give an immediate summary result.
The main feature are:

- Precise rate control (allows to define and change over time the rate at which requests are submitted)
- Precise transactions mixture control (allows to define and change over time % of each transaction type)
- Access Distribution control (allows to emulate evolving hot-spots, temporal skew, etc..)
- Support trace-based execution (ideal to handle real data)
- Extensible design
- Support for statistics collection (microseconds latency and throughput precision, seconds precision for OS resource utilization)
- Automatic rendering via JavaScript (and many available and gnuplot matlab scripts)
- Elegant management of SQL Dialect translations (to target various DBMSes)
- Store-Procedure friendly architecture

This tools is bundled with ten Workloads that all differ in complexity and system demands. Given the type of DBMSes chosen and given the typology of application that will be performed on them, two of ten workloads were selected: *Epinios workload* and *TPC workload*.

Epinions workload

This benchmark is inspired from Epinions.com, that is a customer review website. This is based on previous study [MA05], and uses the data collected with additional statistics extracted from the website. Epinions workload can be similar to many others website workload because it is centered around users interaction. It have nine different tables with twenty-one columns and two primary key and ten indexes. The total transaction that has is nine and it's composed as show on table 3.1. Only update are executed by Epinions workload.
The nine transactions, four interact only with user records, four interact only with item records, and one that interacts with all of the tables in the database. Users have both an n-to-n relationship with items (i.e., representing user reviews and ratings of items) and

Table 3.1: Epinions workload transaction.

Transaction Type	#Select	#Upd/Del/Ins	#Join	Weights	%Read
GetReviewItemById	1	0	1	16	16
GetReviewsByUser	1	0	1	16	16
GetAverageRatingByTrustedUser	1	0	1	16	16
GetItemAverageRating	1	0	0	16	16
GetItemReviewsByTrustedUser	2	0	0	16	16
UpdateUserName	0	1	0	5	0
UpdateItemTitle	0	1	0	5	0
UpdateReviewRating	0	1	0	5	0
UpdateTrustRating	0	1	0	5	0
Total	6	4	3	100	80.00

an n-to-n relationship with users (i.e., indicating a unidirectional ”trust”).

This workload has other three parameters that can be set: the time of execution, the rate and the number of connections that are generated. These parameters have been set respectively to 30 minutes, 10.000 and 100.

Due to the fact that our goal is to find a solution for database used from the website like web-portal, the rate of previous category is balanced to have 80% of read and the rest write.

TPC-C workload

TCP-C workload is the world common workload used to test the DBMS. It is the current industry standard for evaluating the performance of OLTP systems. This benchmark simulates systems working with large volumes of data and execute complex queries.
At low level it consists of nine tables and five procedures that simulate a warehouse-centric order processing application. All of the transactions in TPC-C provide a warehouse id as an input parameter that is the ancestral foreign key for all but one of TPC-C's tables.
The number of New-Order transactions executed per second (for a fixed mixture dictated by the specification) is often used as the canonical measurement for the throughput of a DBMS. One interesting aspect of TPC-C is that if the number of warehouses in the database is sufficiently small, then the DBMS will likely become lock-bound. TPC-C version implemented by Oltpbenchmark is said by developer *a good faith implementation, although we ignore the thinking time requirement for the workers*. This means that each worker issues transactions without pausing, and thus only a small number of par-

Table 3.2: TPC-C workload transaction.

Transaction Type	#Select	#Upd/Del/Ins	#Join	Weights	%Read
NewOrder	4	4	1	0.14	7
Payment	5	5	0	0.15	7.5
OrderStatus	4	0	0	0.31	31
Delivery	3	4	0	0.9	3.85
StockLevel	1	0	1	0.31	31
Total	17	13	2	100	80.35

allel connections are needed to saturate the DBMS. This mitigates the need to increase the size of the database in order to increase the number of concurrent transactions (In the official version of TPC-C, each worker acts on behalf of a single customer account, which is associated to a warehouse).
This workload, is composed by nine tables with ninety-two total columns. The primary key presents is eight and the Foreign key are twenty-four with three different index. The total transaction that has is five and are grouped in category on table 3.2.

The others parameters that can be set on this workload are: the time of execution, the rate, the number of warehouse and the number of connections that are generated. These parameters have been set respectively to 30 minutes, 10.000 , 8 and 100.

As for the previous workload the rate of these transaction are balanced to have 80% of read and the 20% of write.

4

Experiments

This chapter contains the descriptions, characteristics and the methods followed for the experiments. In addition, we will explain the reasons of multiple experiments performed and the results will be examined.
How exposed in previous chapter the workload used are TPC-C and Epinions and for each workload several different benchmark are conducted. One of the results that these tests should highlight is the overhead of each solution and define the base case.
The base case is defined as a solution without a balancer and without a cluster mode, meaning that we'll have a single DBMS powered-on in stand-lone mode. The same typology of tests are executed for each DBMS, even if the types of balancer are different. Any of the preformed tests have the duration of 30 minutes and between any trial there is a delay of 5 minutes. These parameters was chosen by the experience gained in the first round of tests; we have noticed that if we conduct the trials without an adequate delay from each others, a test can start with initial conditions different to the previous one. The duration time was chosen to give a sufficient database stress. At the beginning of this thesis there were two choices: run several test with minor duration time or run one test with a long duration time. After the exploitative trial, we noticed that the first choice's results have a wide gap range between each others; instead, the second choice's results have practically the same range.

The results of each DBMSes test will be exposed on a table like 4.1 that shows the throughput (request/second) for any given solution. The columns *mode* indicate which actors are involved in the test: the values *n1* and *n2* are respectively the first and the second DBMSes node, while the third value indicate the balancer used. If the second DBMS is inside parentheses means that it is synchronized with the first node but without any input connection even if behind a load-balancer: all benchmarks input connections are served directly by the first DBMS node. The others columns contains the results of the test.
At the end of any test is presented a further table that contains the percentage of gain/overhead against the best case.

Table 4.1: DBMS results example

Workload				
mode	I/O noop		I/O cfq	
	swappiness=0	swappiness=60	swappiness=0	swappiness=60
n1	—	—	—	—
n1 + (n2)	—	—	—	—
n1 + n2 + glb	—	—	—	—
n1 + (n2) + glb	—	—	—	—
n1 + n2 + haproxy	—	—	—	—
n1 + (n2) + haproxy	—	—	—	—

This chapter explain also how the load-balancer software used on this thesis was configured and which are the parameters used.
Below the description of any load-balancer and of any DBMSes benchmark.

4.1 Load-Balancing setting

How we stated in previous chapter, the performance of the synchronous solutions can be tested using an load-balancing software. These softwares are open-source, so the installation can be done as for the DBMSes. It is necessary to give a description about how to replicate the experiment done, so the configurations are reported following. Also is explains the reason about the parameters setting for each load-balancing software.

4.1.1 HAProxy configuration

The balancer configurations obtainable with HAProxy are multiple. The HAProxy version used is the 1.4 and it is a stable version. It is possible manage different policies at the same time, such as the possibility to create a chain of front-end or back-end having weight equal or different. Each link of the chain can have different policy of load: it is possible to set a round-robin policy or set a manual priority to any component. All of these characteristics allow to create different types of solutions in high-availability.
The configuration used on this essay has the aim to balance the TPC-C connection in round-robin and to obtain the maximal throughput. The configuration A.1 has been created to achieve this goal; any new connection is managed by the policy stanza *db_write*. It describes the address bind and his behavior: in this case any connection is directed to the stanza *cluster_mariadb* that contains the back-end policy; it describes which are the address where the connections must be balanced, their weight and the policy to do it.
As we can see from the configuration A.1, there is a peculiar setting *option mysql-check*

user haproxy: it defines how to check the life of any node, while the option *check* on the line of each node enables this control; this option is fundamental to achieve an high-availability solution, because if one node dies the balancer must know it immediately to avoid that the new connections are forwarded to the broken node.
Another peculiar setting on the configuration file establishes that each node has the weight equal to one: it means that the load is equally distributed.

4.1.2 GLB configuration

The *Galera load balancer* is a new entry among the Linux balancer. This software is categorized by the developer as beta and we used the 1.0.1 version.
Unlinke HAProxy, Galera load balancer has less options such as the balancing of the HTTP connection: the main purpose of this balancer is to obtain high performances hence it is optimized to work with the Galera solutions, like MariaDB or MySQL.
The settings used are the same ones of the HAProxy: TPC-C connections balanced in round-robin manner, aiming to achieve the maximum throughput. The parameters A.2 are fewer than the previous solution, but there is one that allows to increase the performance: it is the *threads* parameter, and we have set it to 8 after the exploitative tests that is the optimal for the host we are using.
To check if the back-end nodes are alive or dead, it is necessary to use an additional script A.3: it tries a connection with the back-end and, if it fails, the node is considered broken. As for the previous one, this feature is used to execute the tests that were aimed at the calculation of the overhead as a balancer.

4.1.3 PgPool-II configuration

The only balancer that support the load-balancing for PostgreSQL is *PgPool-II*. The version tested is the 3.2.7 and it is the first one available for PostgreSQL 9.3 . The installation package contains the balancer (PgPool-II) and the web-gui (PgPoolAdmin). As we have described in the previous chapter, this balancer has many features but for this essay are just used the load-balancing mode and the connection cache.
Considering that PostgreSQL permits a cluster solution when one node is in read-write mode and the other nodes are in read-only mode, PgPool-II allows to define multiple nodes while the traffic is divided according to the characteristics of each node. As previously stated about the other load-balancer, the goal of this software is to balance the TPC-C connection in round-robin manner and to obtain the maximum throughput; as we can see on the configuration file A.4, the nodes are defined with the same weight. The feature that permits the liveness check is enabled by default, but the automatically fail-over is disable on this tests.
After the exploitative tests, the default option that enables the memory cache was disabled because it is incompatible with the benchmark.

4.2 MariaDB benchmark

MariaDB is unique DBMS examined that support a real synchronized replica in active-active architecture. This means it has two node that can be read and wrote at the same time, obviously this let us have an high-availability solution without the use of others tools. Prerequisite to take advantage of this feature is using a *InnoDB* storage engine.
MariaDB solutions has been tested with 24 different tests. These are organized as following: the first part of tests is needed to understand what is the impact factor of each type of balancer (GLD or HAProxy). In fact a balancer could increase or decrease the performance of the system. To figure out which solution to prefer, for each type of balancer two different tests are conducted: the first is with a balancer between two node, while the second is the utilization of the balancer in single mode, or better with the nodes in cluster mode but the balancer recognizes a single node. In these cases the balancer receive the connection by the benchmark, but in one case makes into a load balancing between the two nodes, while in the other case is a pass-through from the benchmark to one node. These last tests are necessary to understand the balancer overhead.
The second part of tests are executed in stand-lone mode, so the connection from benchmark was done directly to the DBMS without pass through the balancer. With these tests is possible to determine how system's performances change in the presence of a balancer, and if it increase or decrease the throughput of the system. In this phase we have executed two tests: in single node or with two node active in synchronous mode (cluster). With these two tests it is possible to calculate the overhead on a clustered solution and understand which is the best operation system tuning.

The second purpose of this thesis is to estimate the impact of the I/O scheduler and the memory policy. To get these information the tests described below were run four times, and every time was changed one parameter of the machine. In particular we ran 4 different tests changing the following parameter: I/O scheduler (CFQ or NOOP) and memory policy (swappiness=0 or swappiness=60).

For any workload there is a table that shows all results and the best case will be analyzed more carefully.

4.2.1 DBMS configuration

Installation packages *MariaDB Galera Cluster* is trivial and on a *RedHat/CentOS* machine can be done with YUM after that the *MariaDB YUM Repository* is installed. The MariaDB syntax and configuration files are similar to those of MySQL, but some parameters are new. For example the Galera Cluster directives are completely new and to configure a clustered solution these must be correctly set.

Configuration file can be located in */etc/my.cnf* and the settings used in these tests is 4.1.
If a clustered solution is activated, the start-up of the first node differs from the others node because it must initialize the cluster: after that the first node is active, the other nodes can be joined at the cluster in easy way, without any others tricks. The start-up mode consists in using the argument *bootstrap* instead the usual *start*, while to stop any nodes is sufficient to use the argument *stop*.

Listing 4.1: MariaDB configuration's file

```
[mysqld]

# Basic stuff
user                    = mysql
datadir                 = /opt/mariadb
connect_timeout         = 5
max_connections         = 110
wait_timeout            = 100
log-error               = /var/log/mysql.log

# Tuning settings
innodb_buffer_pool_size         = 3372M
innodb_buffer_pool_instances    = 2
innodb_log_file_size            = 128M
innodb_log_files_in_group       = 3
innodb_flush_log_at_trx_commit = 1
innodb_doublewrite              = 1
innodb_file_per_table           = 1
log-slave-updates               = 1
innodb_flush_method             = O_DIRECT
thread_cache_size               = 32
query_cache_size                = 128M
query_cache_limit               = 32M

# Galera Mandatory settings
binlog_format           = ROW
default_storage_engine  = InnoDB
innodb_autoinc_lock_mode = 2
innodb_locks_unsafe_for_binlog = 1

# Galera settings
wsrep_cluster_name = 'galera_cluster'
wsrep_cluster_address = gcomm://192.169.1.1,192.169.1.2
wsrep_provider = /usr/lib64/galera/libgalera_smm.so
wsrep_sst_auth = "sstuser:s3cretPass"
wsrep_node_name = 'dbo01'
wsrep_node_address = 192.169.1.1
wsrep_provider_options = "gcache.size=512M;gcache.page_size=256M"
wsrep_slave_threads = 32
```

```
wsrep_sst_method = xtrabackup
wsrep_auto_increment_control = 1
auto-increment-offset = 1
```

To set a MariaDB DBMS in a clustered solution, we need the add some others options to the configuration file. These have prefix *wsrep* and set up the behavior of the cluster. In particular these one define the name of the cluster (*wsrep_cluster_name*), which can join the cluster (*wsrep_cluster_address*), how much memory is dedicated for the synchronized operation (*wsrep_provider_options*), the number of thread dedicated to the synchronized operation (*wsrep_slave_threads*) and others minor options. Any node have the same parameters except *wsrep_node_name* and *wsrep_node_address* that indicate the name and the ip address of any nodes.
As previously mentioned, Galera Cluster must be used with *Innodb*, so is mandatory to set appropriate option on this storage engine.

When there is a comparison of some tests, it's important that the initial state is identical for each tests. One factor that may affect the results is the initial state of memory; in fact if the DBMS has just started and no operation was done before, the memory is empty and no data is cached. To avoid this problem and to have an identical initial state for each test, an initial spoof test was performed and the benchmark was started in a clustered mode. After that all tests in clustered mode was done, the second node is stopped and the single node test was performed.

4.2.2 TPC-C workload results

The TPC-C workload has heavy-write load: although we have set the workload with a 80% of reading, the overall performances of the system are affected.
How we can see on the table 4.2 there are some interesting results: the first is that the best results are obtained with the *NOOP* scheduler I/O with the memory swap policy to the default value. This result is a logical consequence of scheduler I/O; in fact with the storage layer utilized on this thesis, if the incoming I/O requests are not ordinated then these are more fast. The *swappiness* to 60 contribute to increase the performance because the swap is a dedicated space of the disk and this is subject to the same I/O scheduler. When the page of memory becomes inactive, it is moved to the swap memory but the reference is kept to the page table; after that the page is moved to the swap, if it is necessary is sufficient read this area of memory and is not necessary create new page.
On the other side, the best results with the I/O scheduler CFQ are obtained with the swappiness policy to 0. This result is still a consequence of the I/O scheduler, because the *CFQ* policy utilizes a ordinate queue that has a slower performance than *NOOP*. Since the swap activity utilizes the storage layer, if it is greatly slower it is better to avoid it. This is the reason because the best result with the *CFQ* I/O scheduler was

obtained with the swap policy to 0.
If we consider the column with I/O scheduler to *NOOP* and swappiness to 60, the two best results (in bold type) were obtained with a standalone solution and with a GLB balanced solution. Into all configurations can be noted that the best throughput was obtained with the GLB balancer; this solution among the HAProxy are, on average, slower than 0.7%.

Table 4.2: MariaDB-TPC results

TPC-C Workload				
mode	I/O noop		I/O cfq	
	swappiness=0	swappiness=60	swappiness=0	swappiness=60
n1	1057.8910	**1094.9105**	1080.0982	1064.8587
n1 + (n2)	980.2883	988.0496	910.4753	838.6430
n1 + n2 + glb	802.7463	**1077.3802**	699.1509	574.7270
n1 + (n2) + glb	916.0546	1027.9965	949.3333	929.4890
n1 + n2 + haproxy	760.1499	939.9956	653.9604	571.4477
n1 + (n2) + haproxy	977.2686	985.6206	891.9196	986.1674

The behavior of the system in standalone mode can be seen on the figure 4.2 4.3 4.4 4.5.
As we can observe on the image 4.1 the throughput of the DBMS is quite smooth except for a few negative peaks that are caused by the disk flush. Of course when the throughput drops the latency increase, though the latency average is about 100 ms.
If we consider the memory graphic, the swap memory used is increasing on the first five minutes to stabilize when the total memory usage is constant over the 95%. It is interesting to notice that the cache has more small negative peaks corresponding to the disk flush.
The graph of disk have some write peaks and the average of that is about 15MB/sec, while the read operation consumes less bandwidth, about 5MB/sec.
It is interesting to note that the network bandwidth has instead a major send-activity then the receive, but it can be justified by the topology of workload that has been set with the 80% of read activity.
The CPU activity is affected by the disk activity. In fact the negative peaks correspond to the flush operation, while the normal pattern show that the system is working properly. The lack of idle presence or system activity means that the disk layer is not a bottleneck and the resources of the systems are adequate.

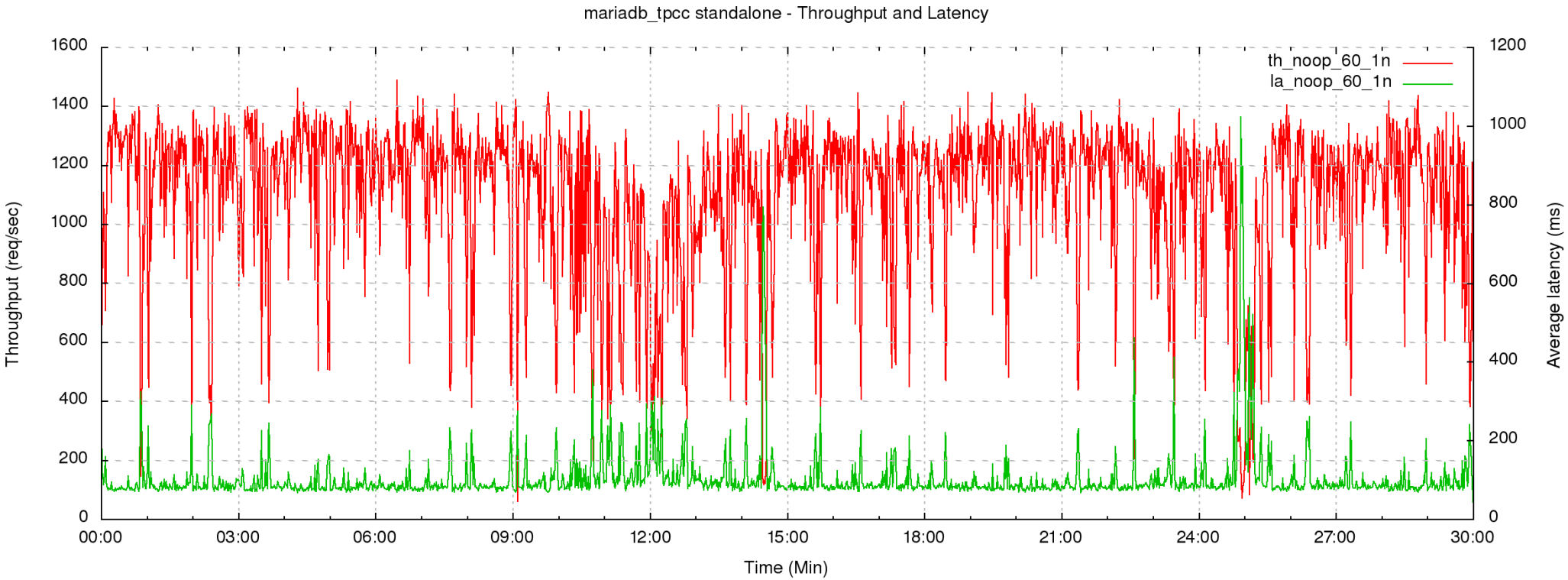
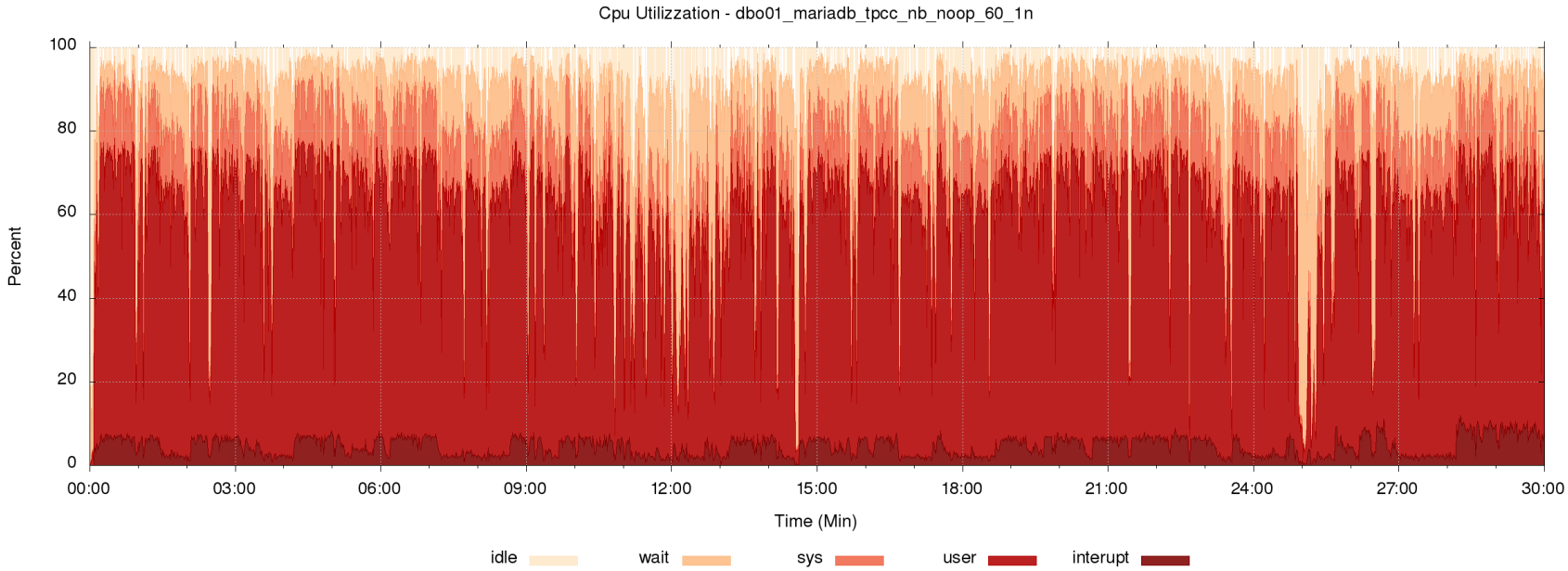


Figure 4.1: MariaDB TPC-C Throughput on a single node without a balancer

Figure 4.2: MariaDB TPC-C benchmark: CPU load of node one in stand-lone mode



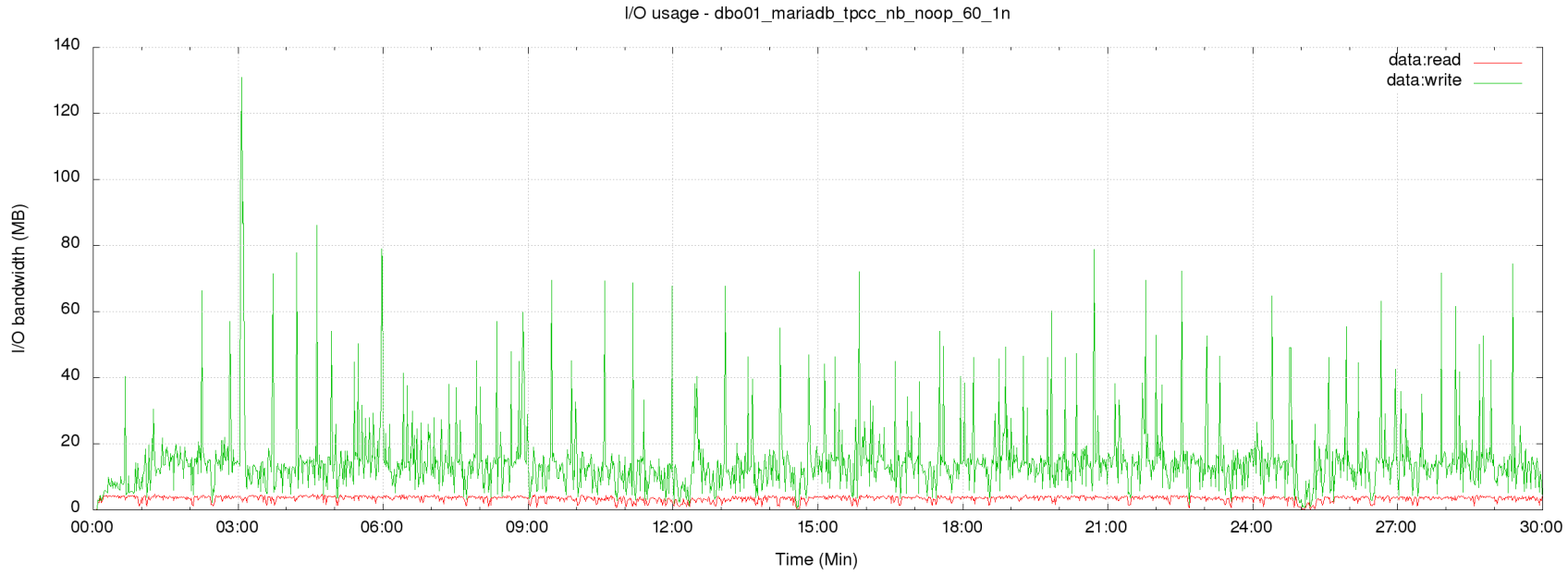
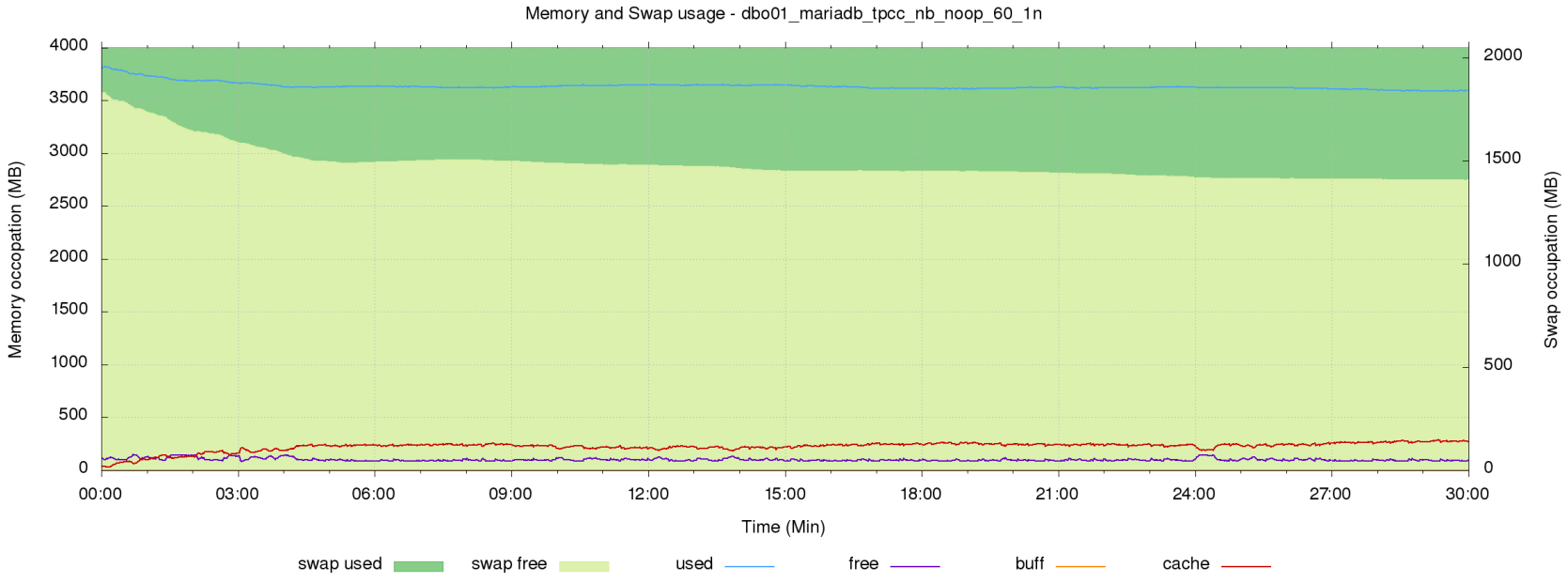


Figure 4.3: MariaDB TPC-C benchmark: disk load of node one in stand-lone mode

Figure 4.4: MariaDB TPC-C benchmark: memory load of node one in stand-lone mode



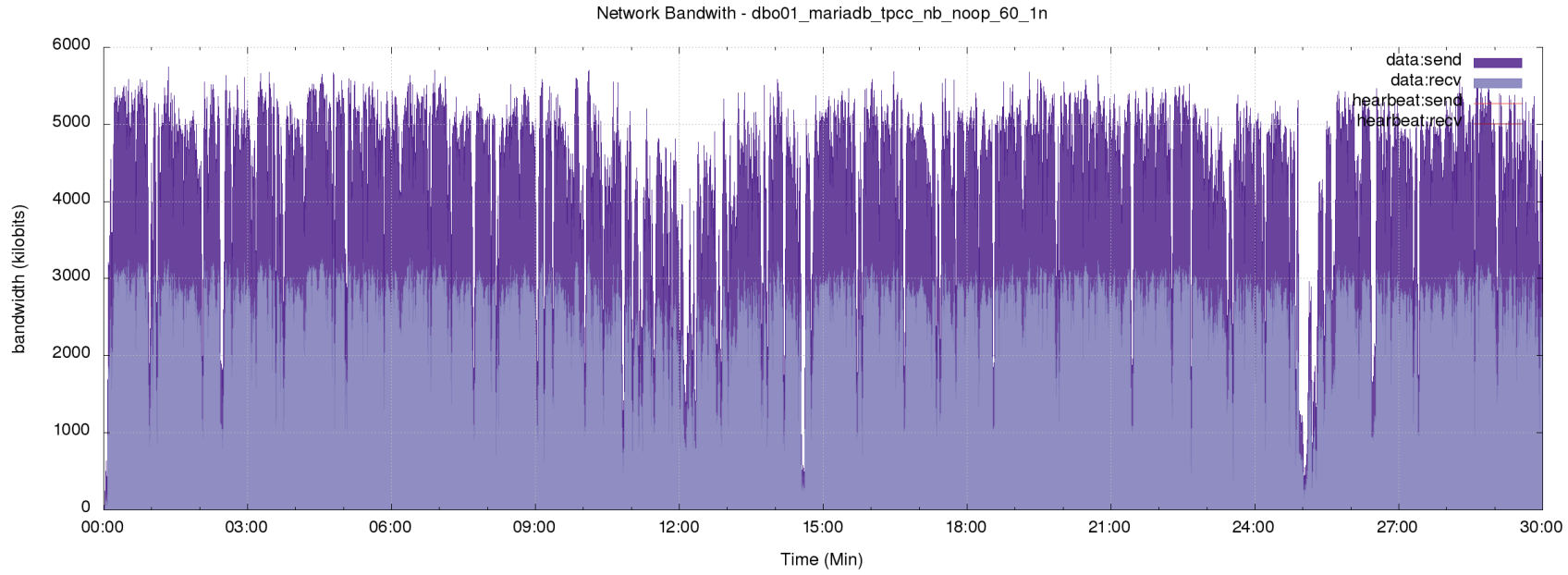


Figure 4.5: MariaDB TPC-C benchmark: network throughput of node one in stand-lone mode

The cluster solution with two nodes on the GLB, has the similar behavior of the standalone configuration.
The figure 4.6 show the throughput of the DBMS, and you can see how the average latency is about 100ms while the negative peaks of throughput correspond to the flush operation on the two nodes. When one peak appears can be caused by the flush operation of the node 1 or node 2. The behavior of the node 1 can be seen on the figure 4.7 4.8 4.9 4.10 while for the node two on the figure 4.11 4.12 4.13 4.14.
It is interesting to notice that the CPU load of the two nodes has a major percentage of idle then the standalone solution: it corresponds to a minor user load because the percent of the system and system itself are the same.
The disk activity of the two nodes has the same bandwidth over the standalone solution. This means that the disk is not a bottleneck for the cluster solution.
Even in this case, the memory has the same behavior of the standalone solution: the swap free decrease for the first 5 minutes, while the memory usage is constant all over the experiment. There are some small peaks in the cache line: these are caused by the flush of the memory over the disk.
An interesting aspect is the network graph. The bandwidth to and from the balancer is divided over the two nodes and when there is a negative peak on node 1 there is a positive peak on the node 2 and vice-versa. The heartbeat bandwidth has the same phenomenon of the data network and the throughput of this network is lower than the physical limit.

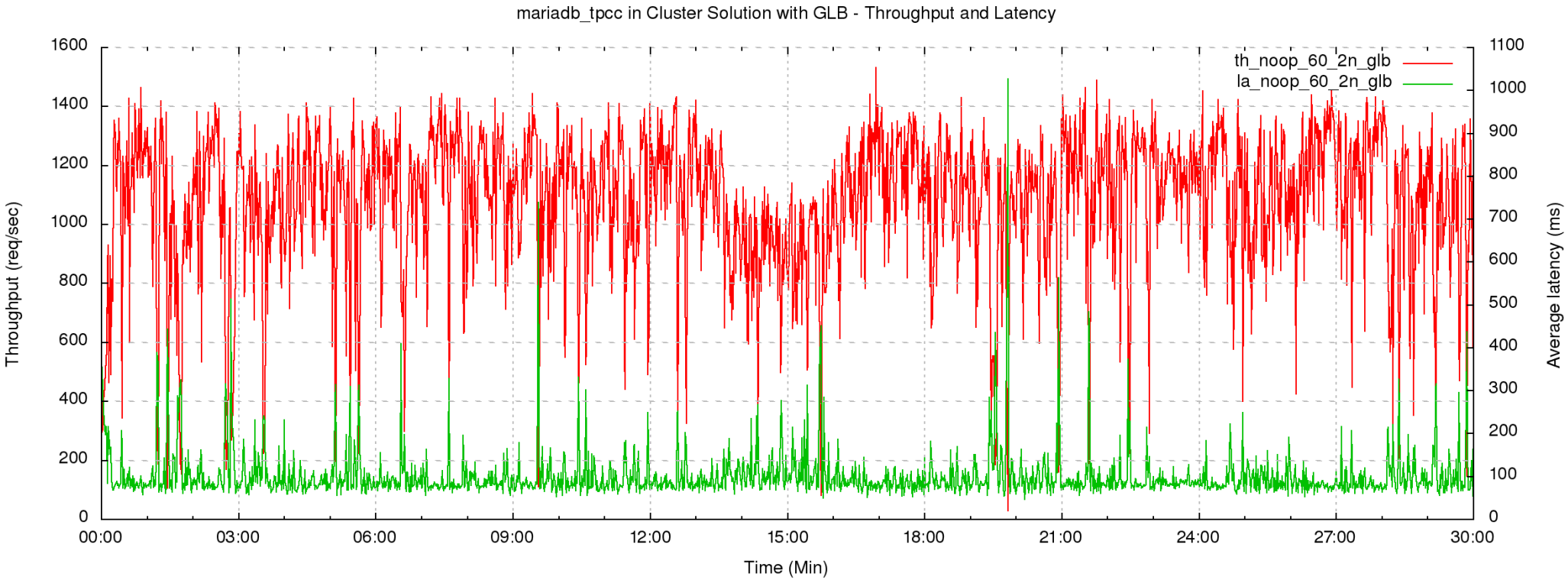
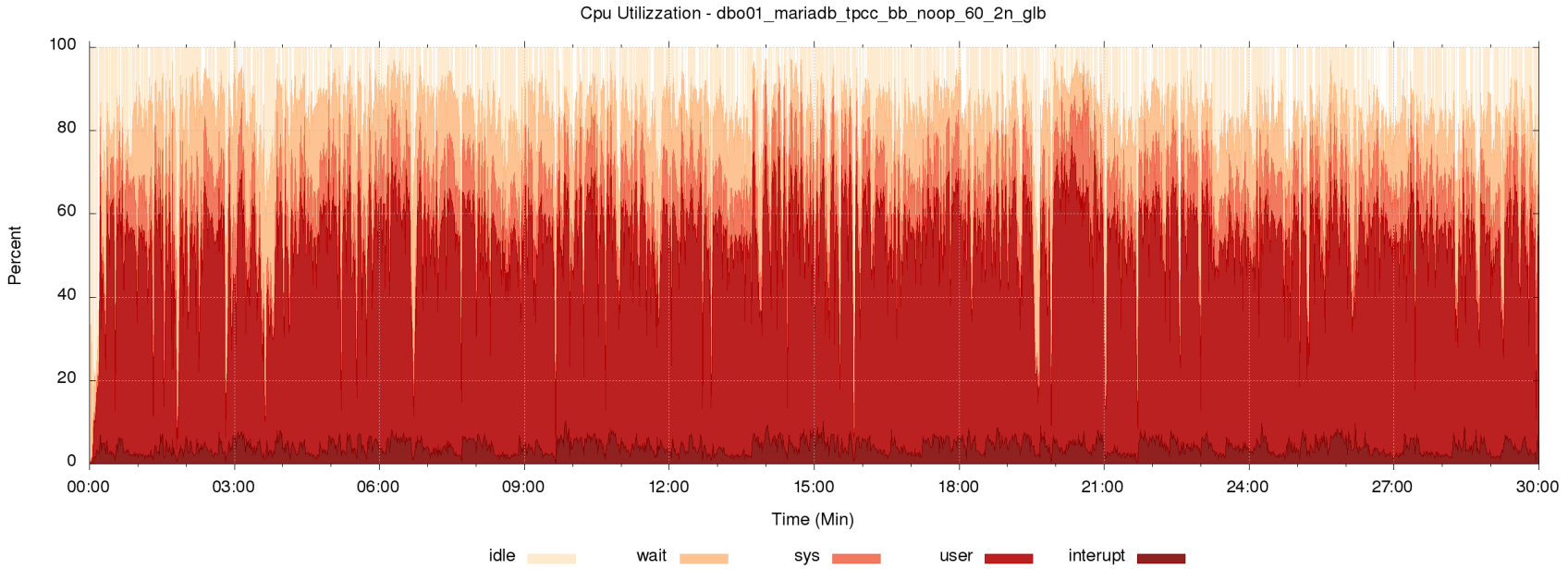


Figure 4.6: Epinions Throughput on a cluster solution with two nodes active and the balancer GLB

Figure 4.7: MariaDB TPC-C benchmark: CPU load of node one in streaming replication with GLB balancer



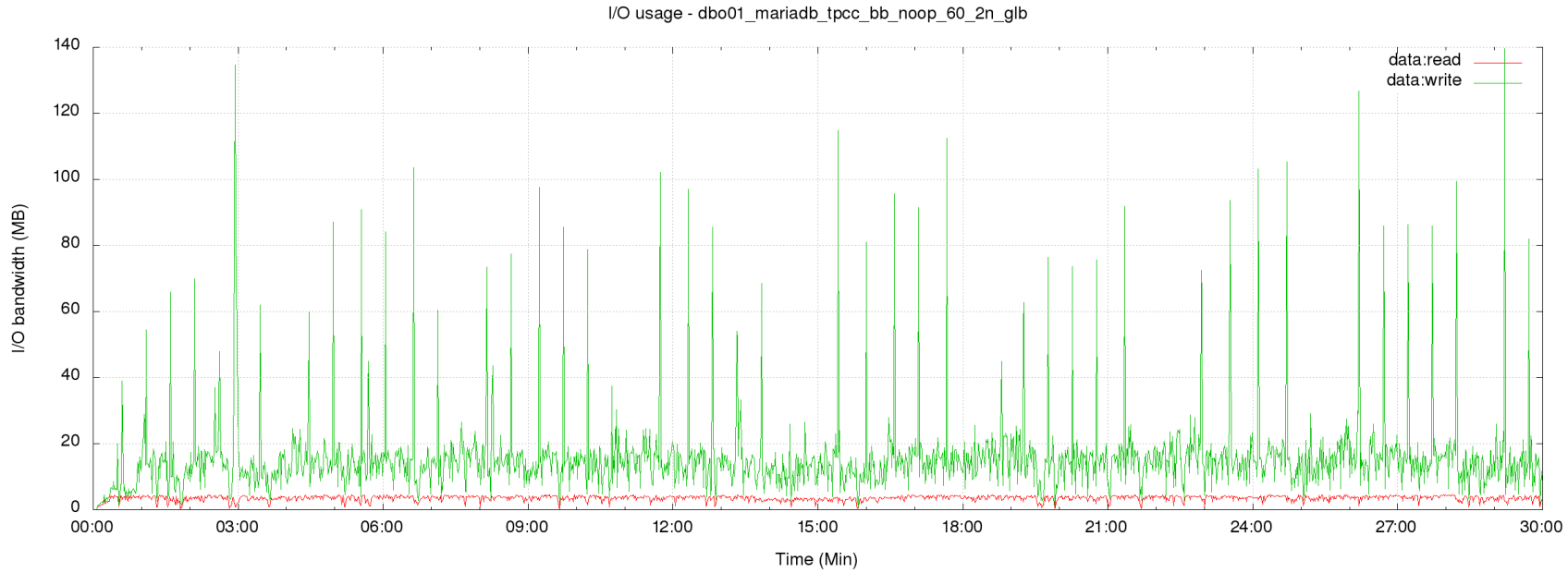
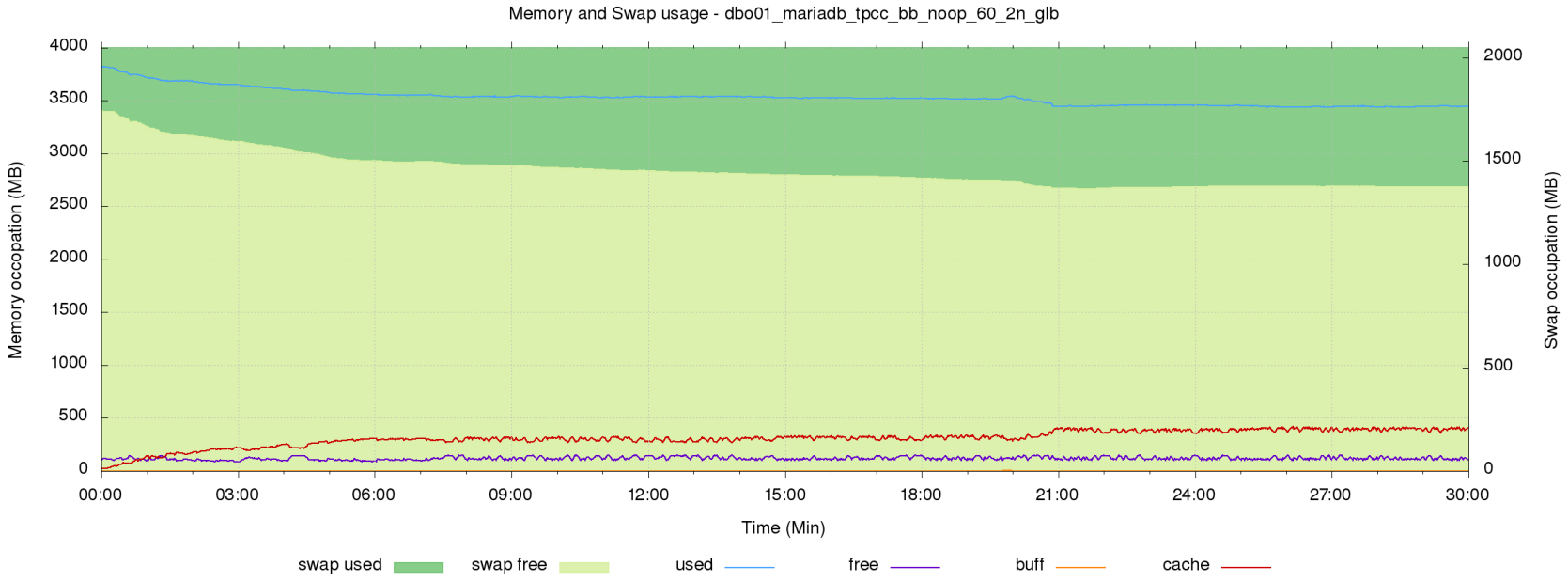


Figure 4.8: MariaDB TPC-C benchmark: disk load of node one in streaming replication with GLB balancer

Figure 4.9: MariaDB TPC-C benchmark: memory and swap load of node one in streaming replication with GLB balancer



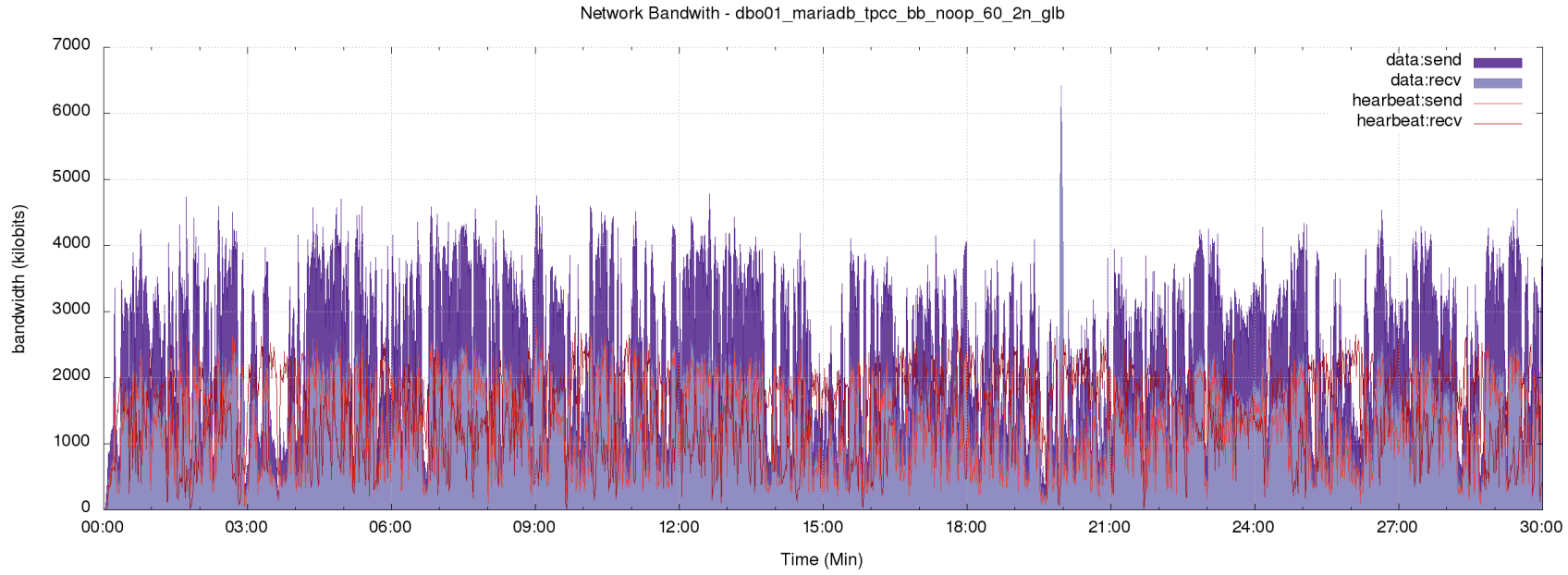
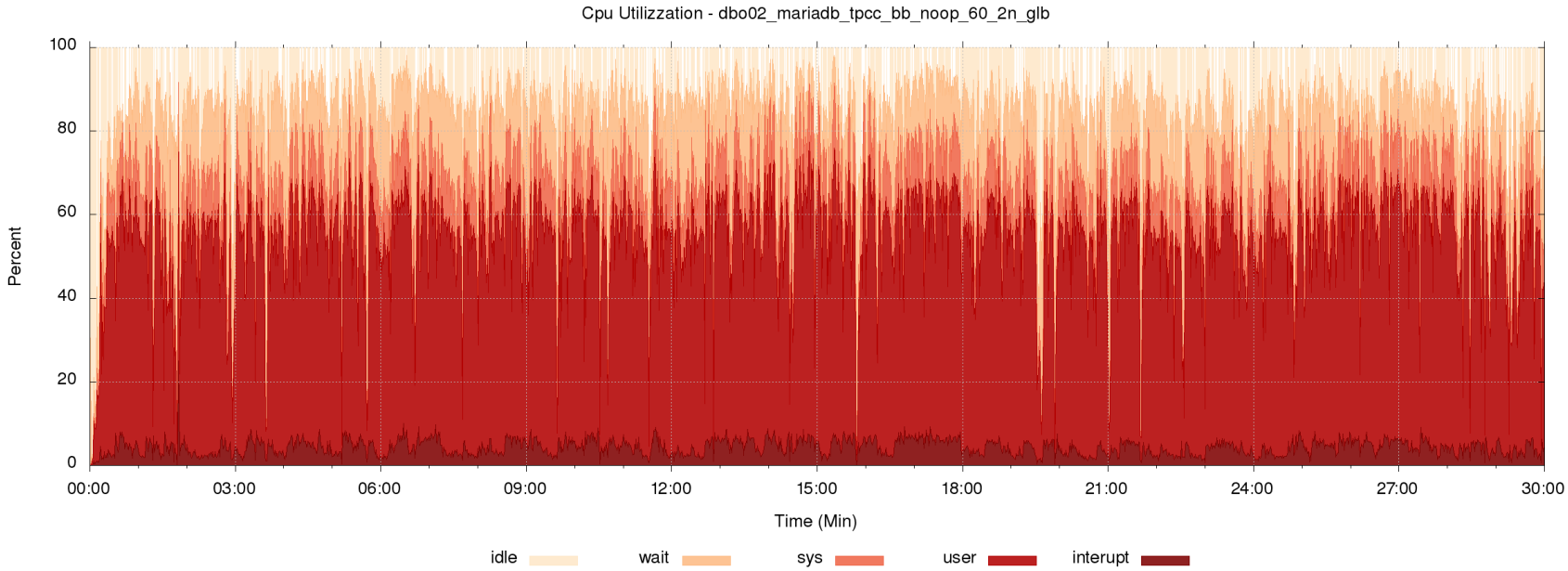


Figure 4.10: MariaDB TPC-C benchmark: network throughput of node one in streaming replication with GLB balancer

Figure 4.11: MariaDB TPC-C benchmark: CPU load of node two in streaming replication with GLB balancer



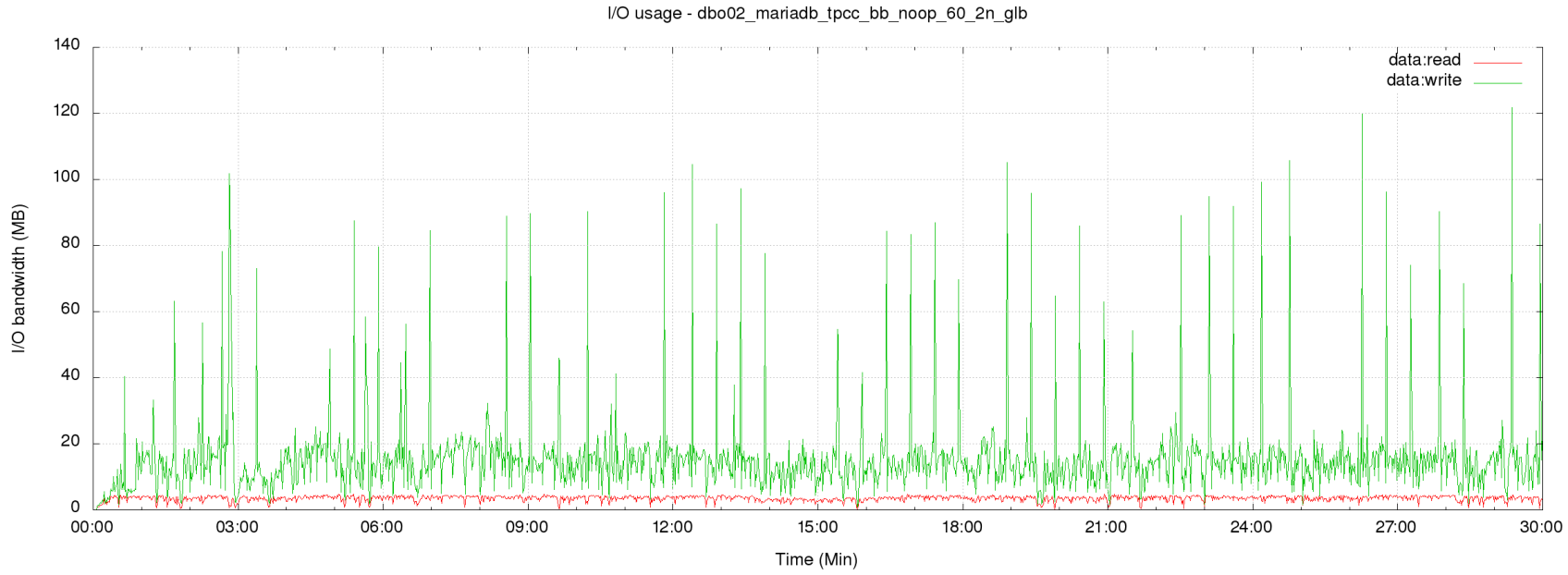
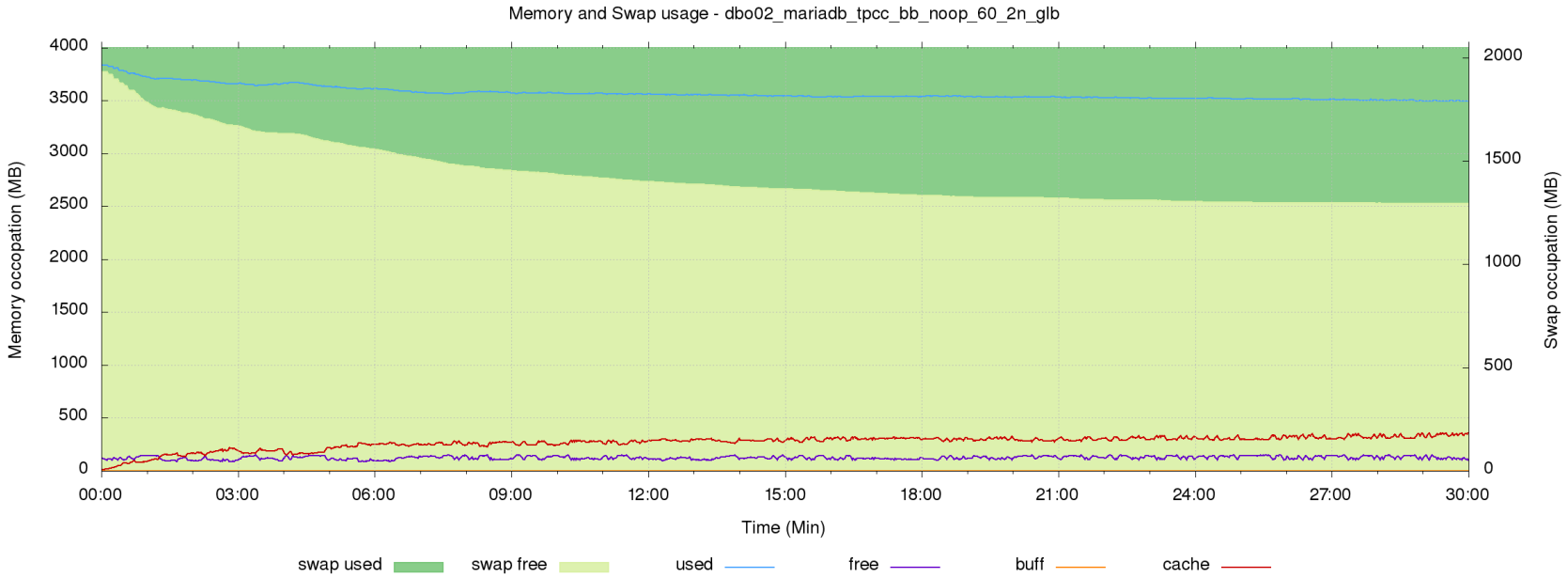


Figure 4.12: MariaDB TPC-C benchmark: disk load of node two in streaming replication with GLB balancer

Figure 4.13: MariaDB TPC-C benchmark: memory and swap load of node two in streaming replication with GLB balancer



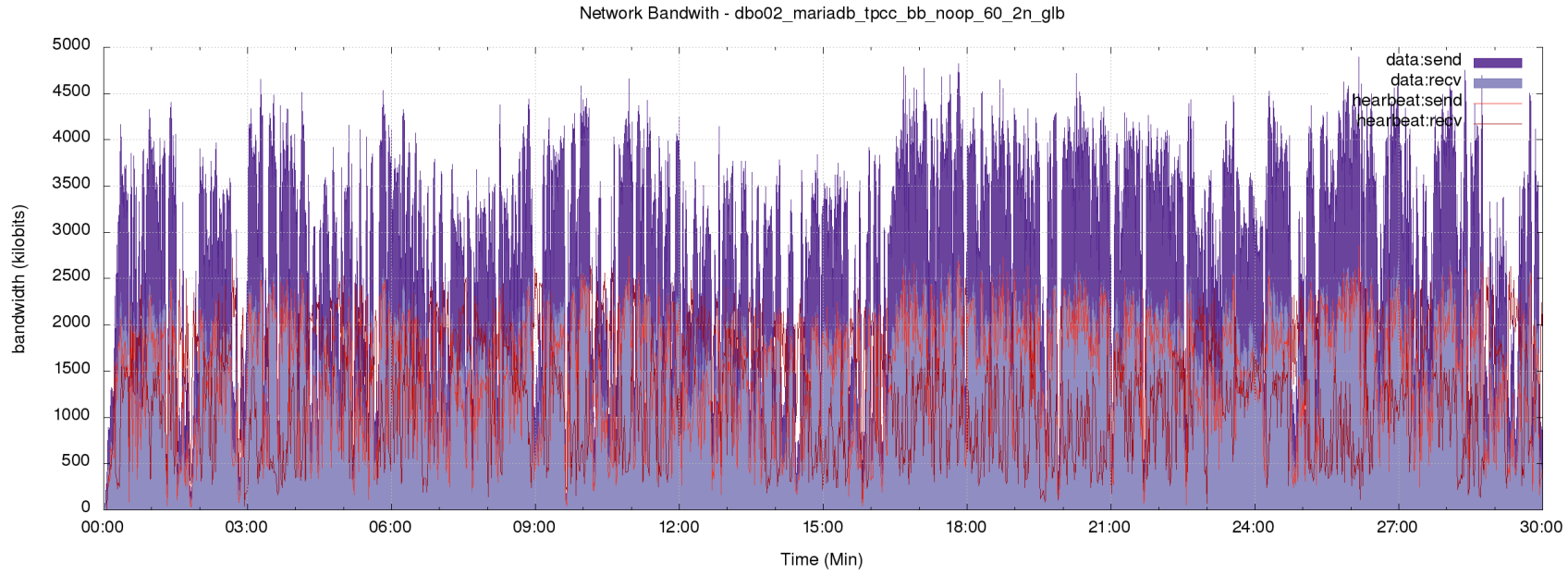


Figure 4.14: MariaDB TPC-C benchmark: network throughput of node two in streaming replication with GLB balancer

4.2.3 Epinions workload results

The Epinions workload has a load similar to a common web portal, that means that the nine transactions have not a high write's intensity. By the table 4.3, you can see some interesting results. The foremost result we have obtained is that the two best configurations are with the scheduler I/O set to *NOOP* or cfq and the swappiness set to 0. The reason is due again to the type of storage layer used. In this case the DBMS does not have a high number of operations read/write because the memory has sufficient capacity to contain the entire database, and it is obvious that a policy that avoid the swap is the better choice because all memory pages are inside the RAM.
The other result that the table 4.3 shows is that the better performance is obtained with the solution in cluster among the GLB. Unlike the TPC-C workload where the standalone solution had the higher raking, the synchronous solution with the GLB balancer had a 13.84% gain than the standalone solution. This result is possible if it is used the scheduler I/O *NOOP*, while with the scheduler I/O *CFQ* the better solution is the standalone.
If you compare the solution that use the two balancer, an other clear result is that HAProxy has poorer performance than GLB, and the average overhead about these solution is the 0.04%.

Table 4.3: MariaDB-Epinions results

Epinions Workload				
mode	I/O noop		I/O cfq	
	swappiness=0	swappiness=60	swappiness=0	swappiness=60
n1	**8033.9359**	6578.6001	8013.6395	6928.3714
n1 + (n2)	6201.7492	6190.1735	7300.9570	7231.2530
n1 + n2 + glb	**9143.9628**	8694.1304	6428.4041	4785.7125
n1 + (n2) + glb	7588.1312	6778.5137	6741.9024	6315.0807
n1 + n2 + haproxy	8510.2935	8001.7623	7112.0127	6460.7784
n1 + (n2) + haproxy	7303.0546	7209.6801	6359.3836	6441.9358

The behavior of the system in standalone mode can be seen on the figure 4.16 4.17 4.18 and 4.19.
The image 4.15 illustrates the throughput and the latency of the DBMS. The average of latency values is about 20ms and the peaks are the opposite of the throughput values. These peaks are caused by the disk flush. In fact we can observe negative peak of cache memory and the positive peak of disk write.
If you consider the memory graph, the swap memory is never used because the swappiness value is to 0. There are other two interesting features on this graphics: the first one is that the percent of memory usage is always about 95%, that means that the database

is in the RAM, and the second one is that the frequency of the flush of memory is more frequent while time passes. In fact the trend jagged of the cache and buffer value thins during time, and at the opposite the negative peak of the throughput line are always more frequent and less deep.
The types of transaction that was performed by the workload are such as to have a high number of resulting rows, then it is trivial that the percentage of data sent is greater than those received. The total network bandwidth used is less than the maximum obtainable with a gigabit network, so this is not a bottleneck.
On the CPU graphic, there is a small percentage of wait activity that correspond of the write activity, the percentage of SYS load is constant for the whole duration of the test as well as the percentage of the user load. Overall the system has 10% of idle percentage, so the resources of the systems are sufficient for this workload, and the disk or the network are not a bottleneck.

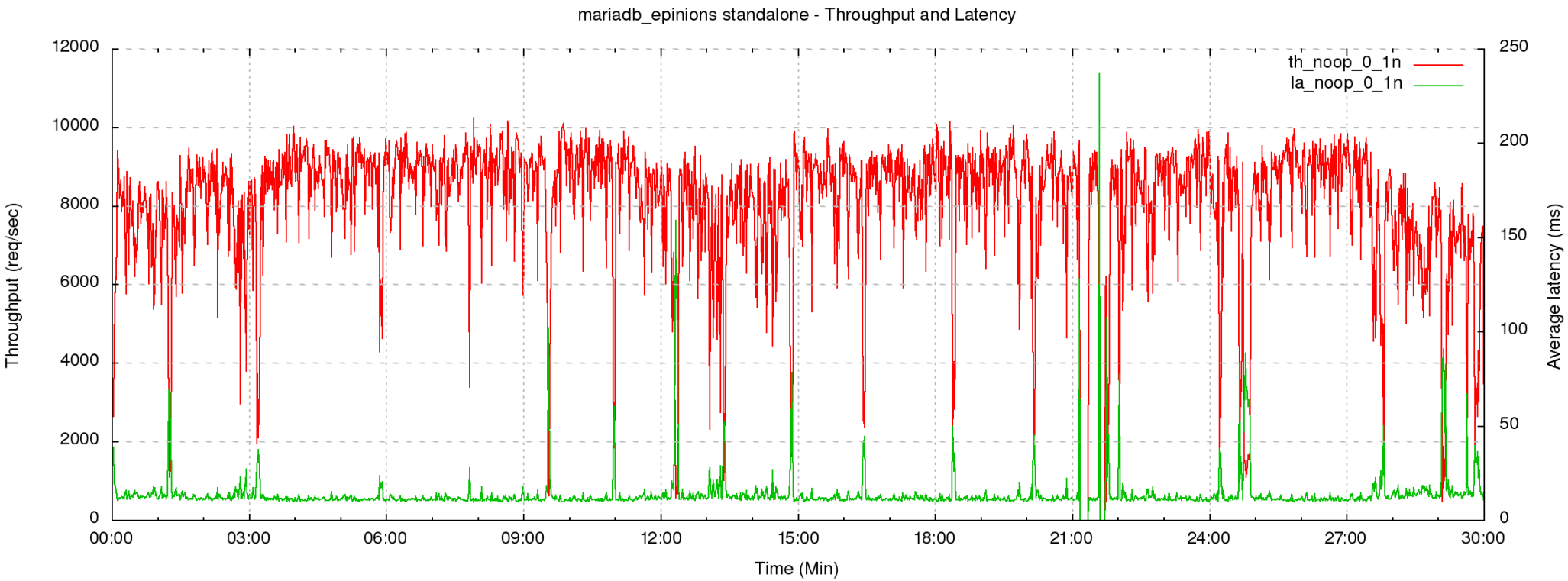
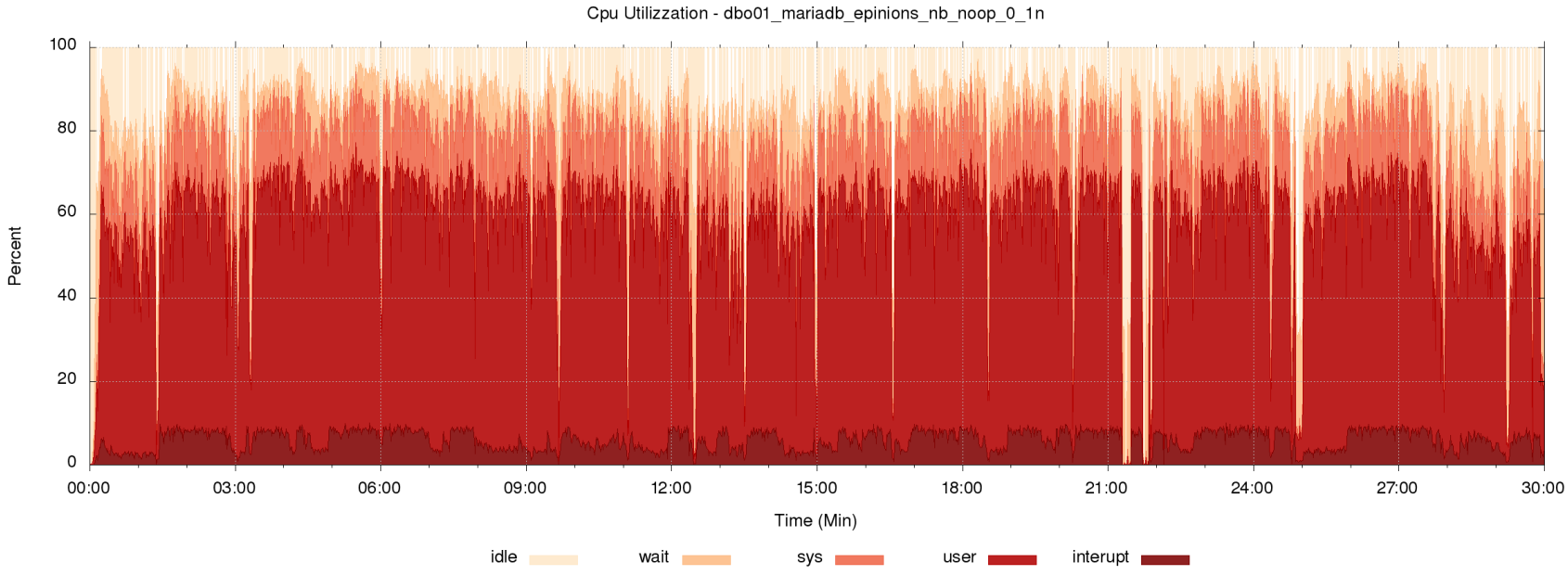


Figure 4.15: MariaDB Epinions throughput on a single node without a balancer

Figure 4.16: MariaDB Epinions benchmark: cpu load of node one in streaming replication with GLB balancer



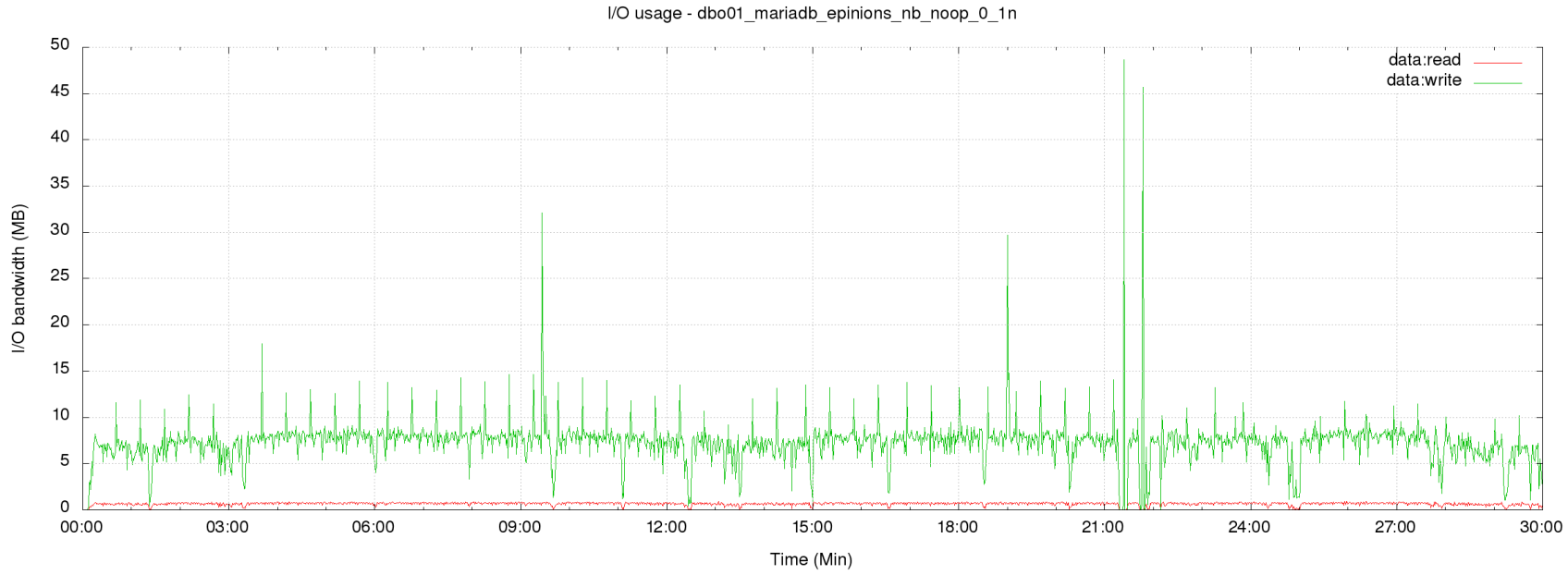
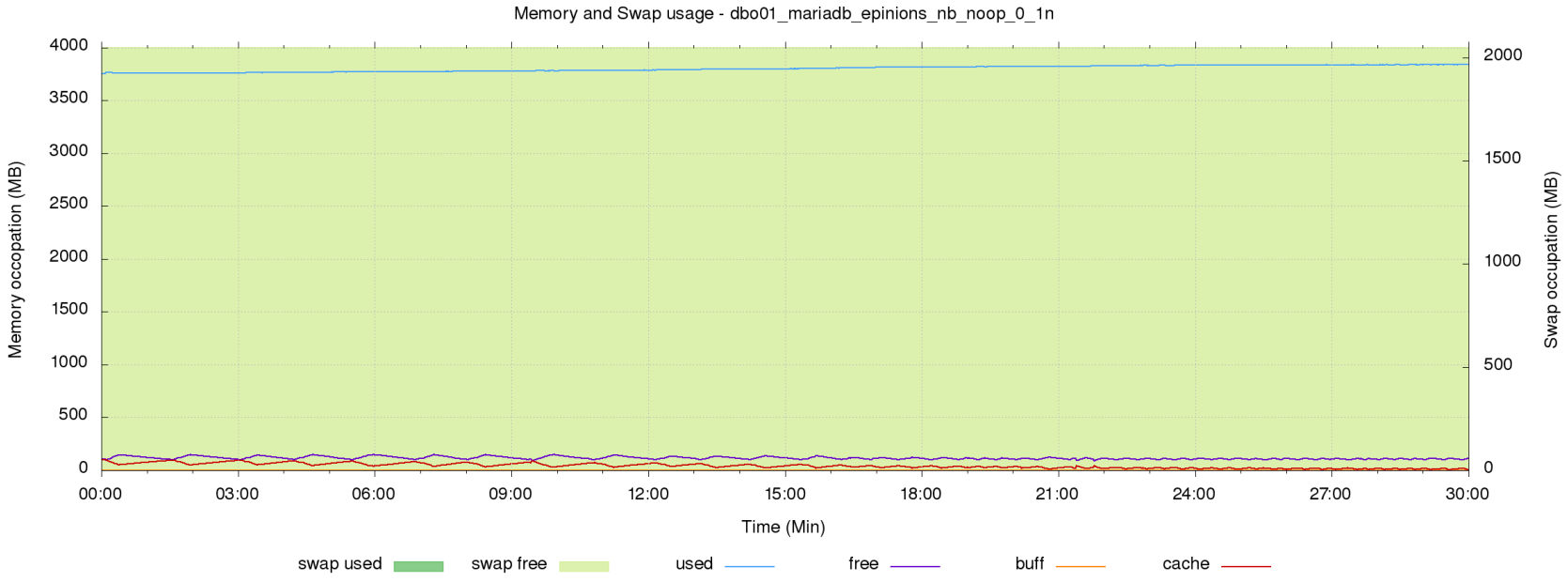


Figure 4.17: MariaDB Epinions benchmark: disk load of node one in streaming replication with GLB balancer

Figure 4.18: MariaDB Epinions benchmark: memory load of node one in streaming replication with GLB balancer



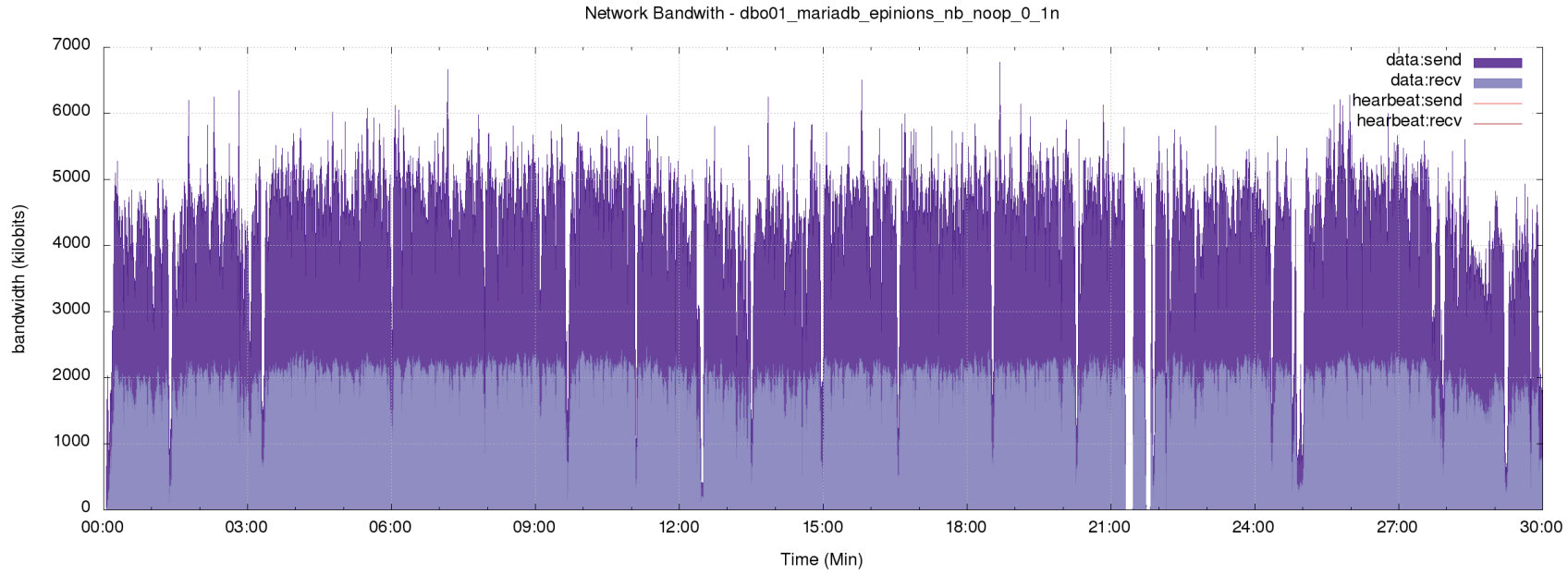


Figure 4.19: MariaDB Epinions benchmark: network throughput of node one in streaming replication with GLB balancer

The Epinions workload on the cluster solution with the balancer GLB involves an almost perfect symmetry in the two systems analyzed.
If we start from the figure 4.20, we can notice that the throughput behavior is almost regular. The negative peak, as you can see from the figure 4.22 and 4.26 that show the behavior of the database disk, are caused by the storage activity. This trend may be defined as normal because it is asymptotic and with duration of 1 second.
The DBMS latency is on average less than 50ms ,apart some cases it corresponds to the negative peak of the throughput. The trend symmetry of the two nodes can be observed over the disk graphs. In fact the negative and positive peaks of the node 1 images, are symmetric over the node 2 images. This behavior is obviously due to the synchronization between nodes.
It is interesting to notice that the CPU load 4.21 4.25 of the two node has on average 23% of percentage idle: it means that the CPU is not a bottleneck otherwise the utilized percent would be close to 100%. Another interesting point is that the percent of WAIT and SYS are constants over all time of the experiment.
On the memory graph 4.23 4.27, we can see that the swap is never utilized by node 1 and node 2. Also the trend of the cache value is constant, although there is a slight increase at the beginning. The behavior of the memory used and free is quite the opposite. Obviously there is a growing trend of memory occupied, and a decreasing trend of free memory. This phenomenon is not located within the end of the 30-minutes test, but it is easy to assume that at the moment of the end of free memory there is a gradual decrease in the cache memory since this is used as a disk optimization. In the event that the cache's space would end and the system would need more memory you could use the swap's space, but this hypothesis would not be feasible because it would go against the very reason of cache memory, a feature that is used to enhance the performance.
The network graphs 4.24 4.28 show the behavior of the data and heartbeat network interfaces. The trend of the data interfaces is the same trend of standalone solution. In fact the percentage of data sent is higher than the received ones because the kind of executed queries returns many rows and much more data. Instead, the heartbeat interface has lower level that data network and it is the opposite of the TPC-C workload. This difference is due to the type of workload because the types of transactions on the Epinions are different towards the TPC-C workload that has heavy-write load.

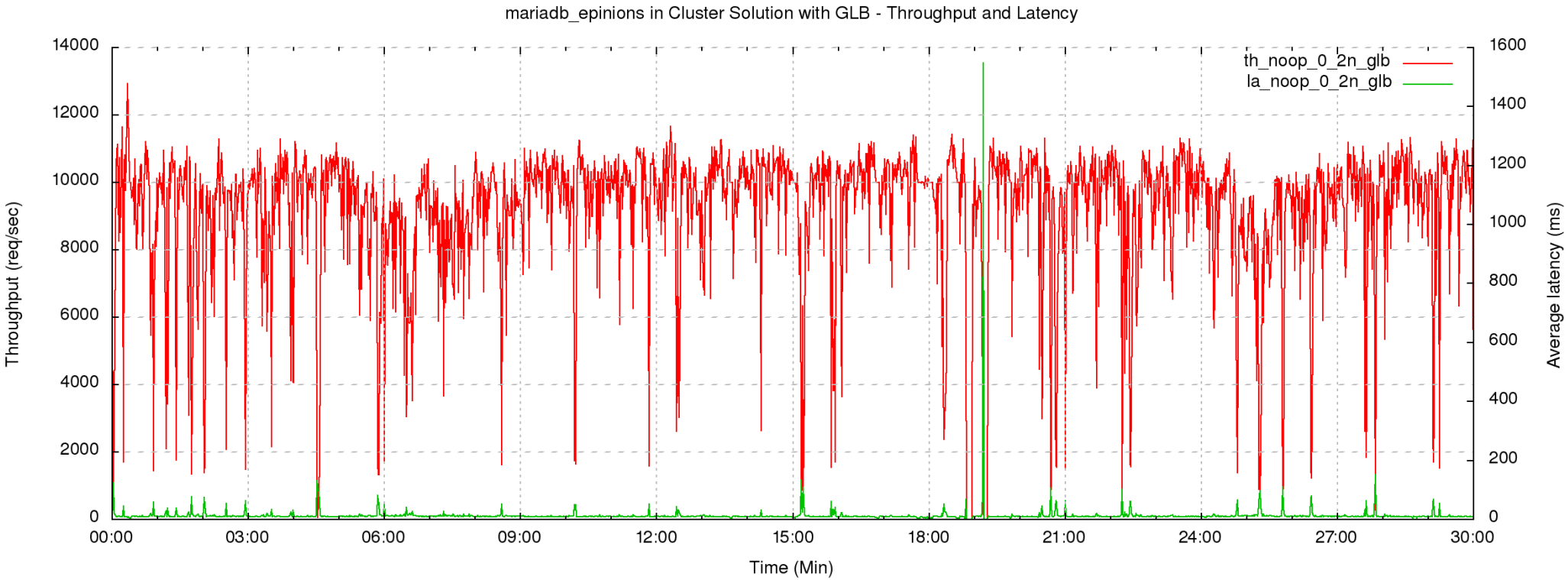
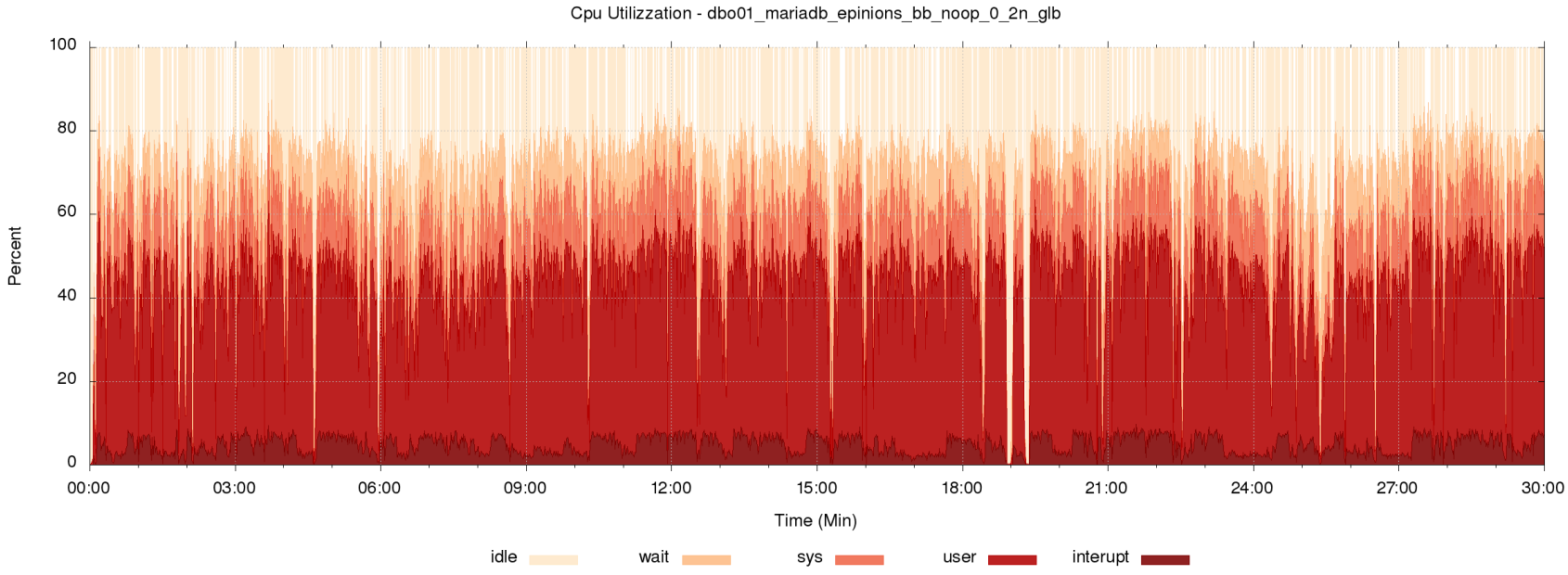


Figure 4.20: Epinions throughput on a cluster solution with two nodes active and the balancer GLB

Figure 4.21: MariaDB Epinions benchmark: cpu load of node one in streaming replication with GLB balancer



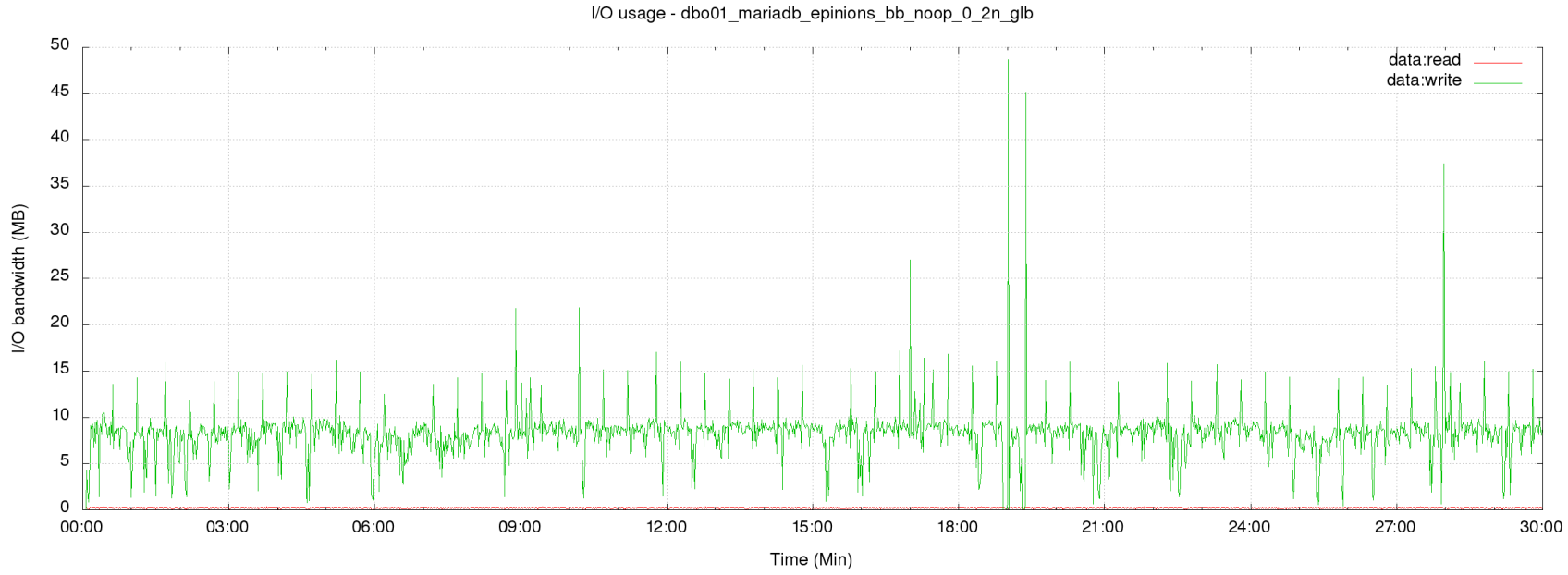
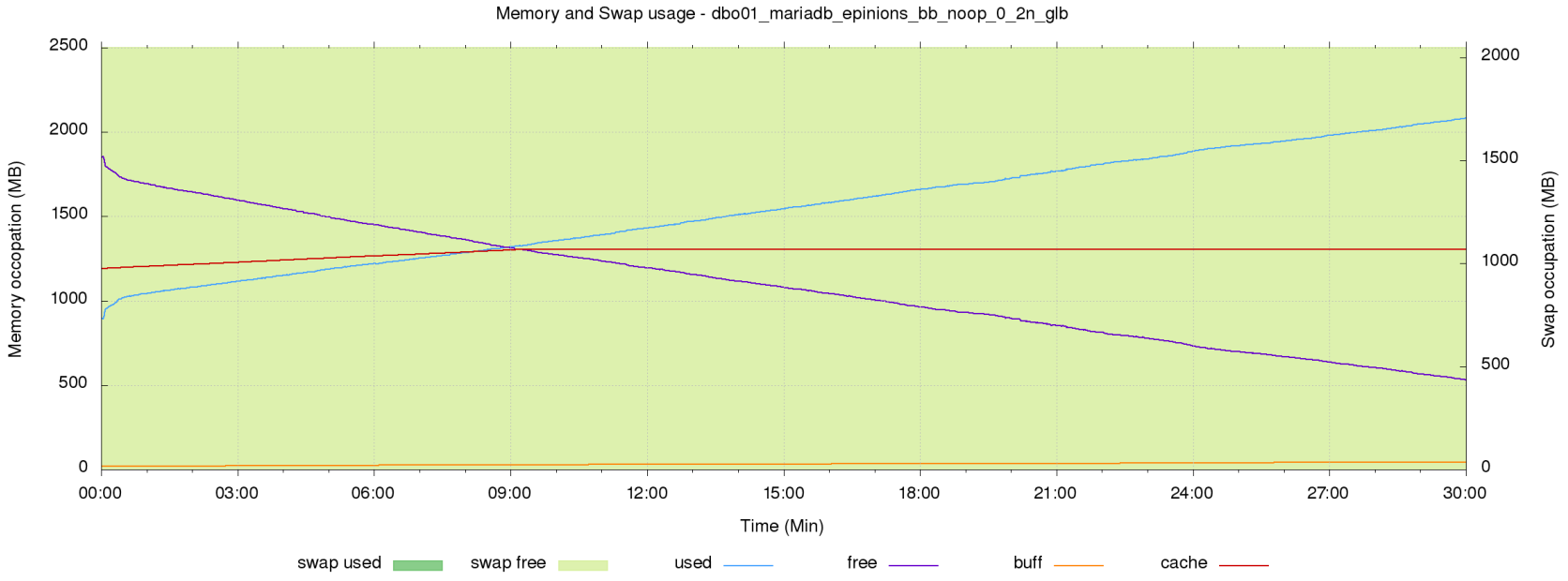


Figure 4.22: MariaDB Epinions benchmark: disk load of node one in streaming replication with GLB balancer

Figure 4.23: MariaDB Epinions benchmark: memory and swap load of node one in streaming replication with GLB balancer



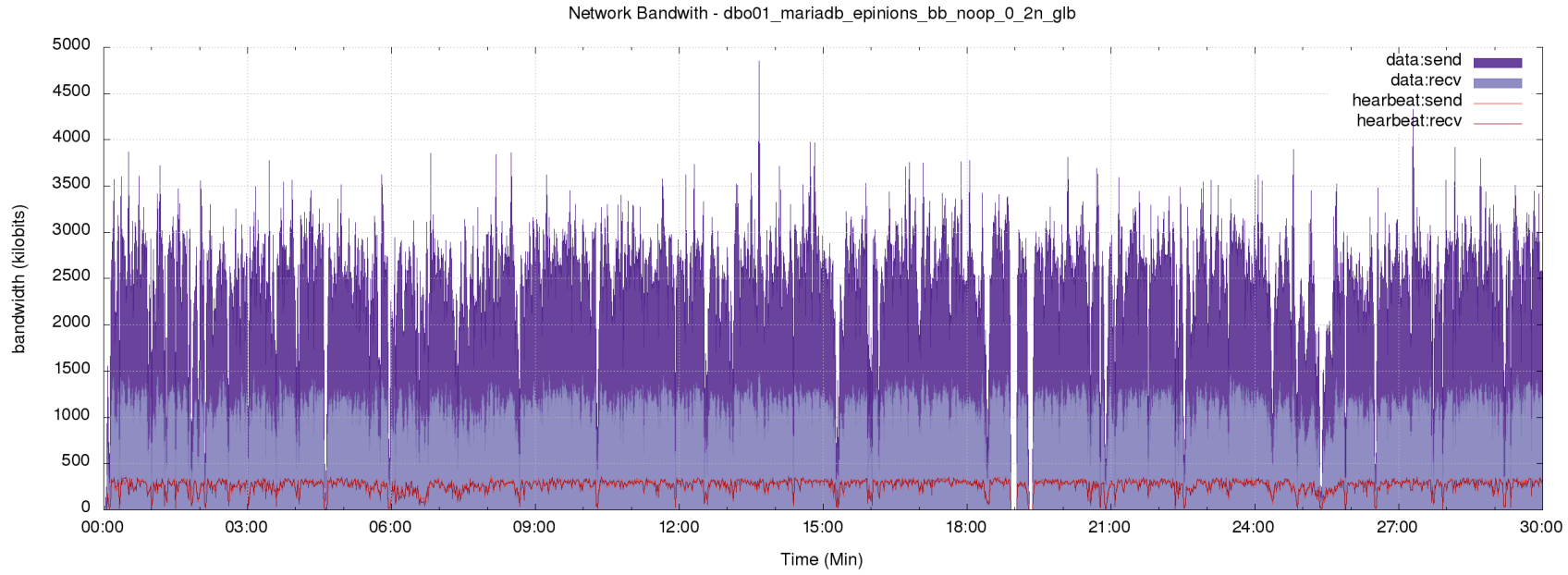
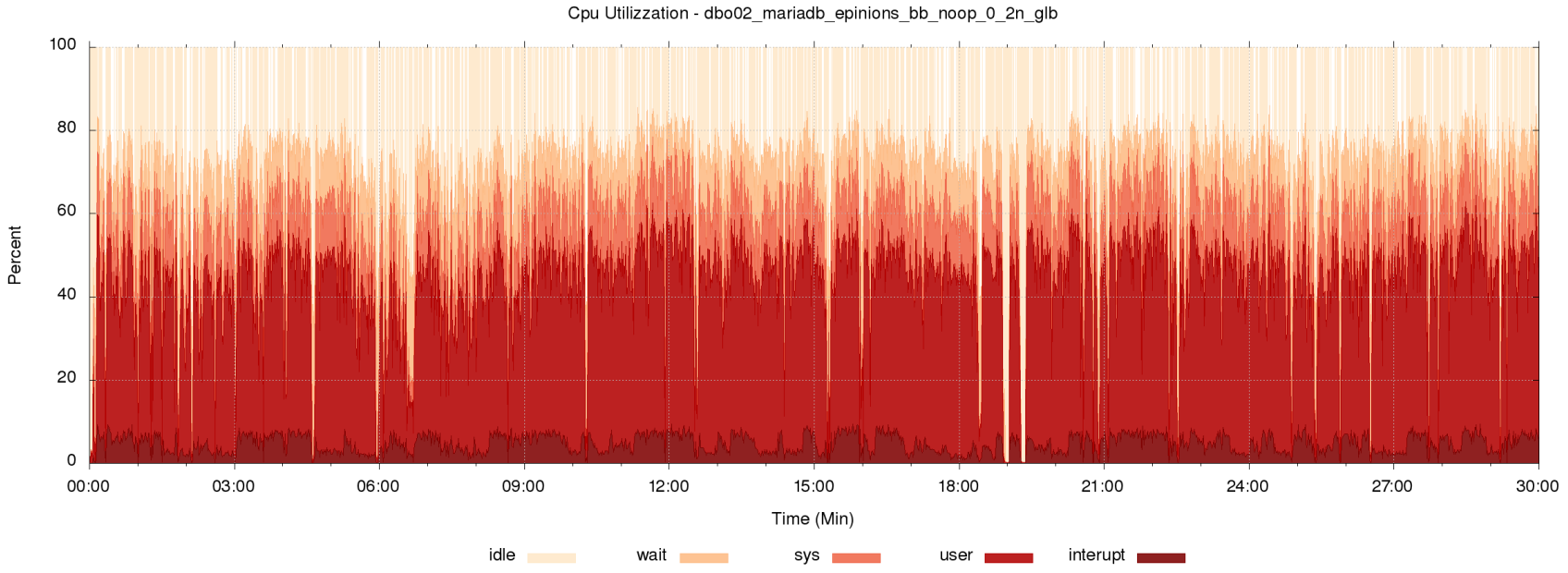


Figure 4.24: MariaDB Epinions benchmark: network throughput of node one in streaming replication with GLB balancer

Figure 4.25: MariaDB Epinions benchmark: cpu load of node two in streaming replication with GLB balancer



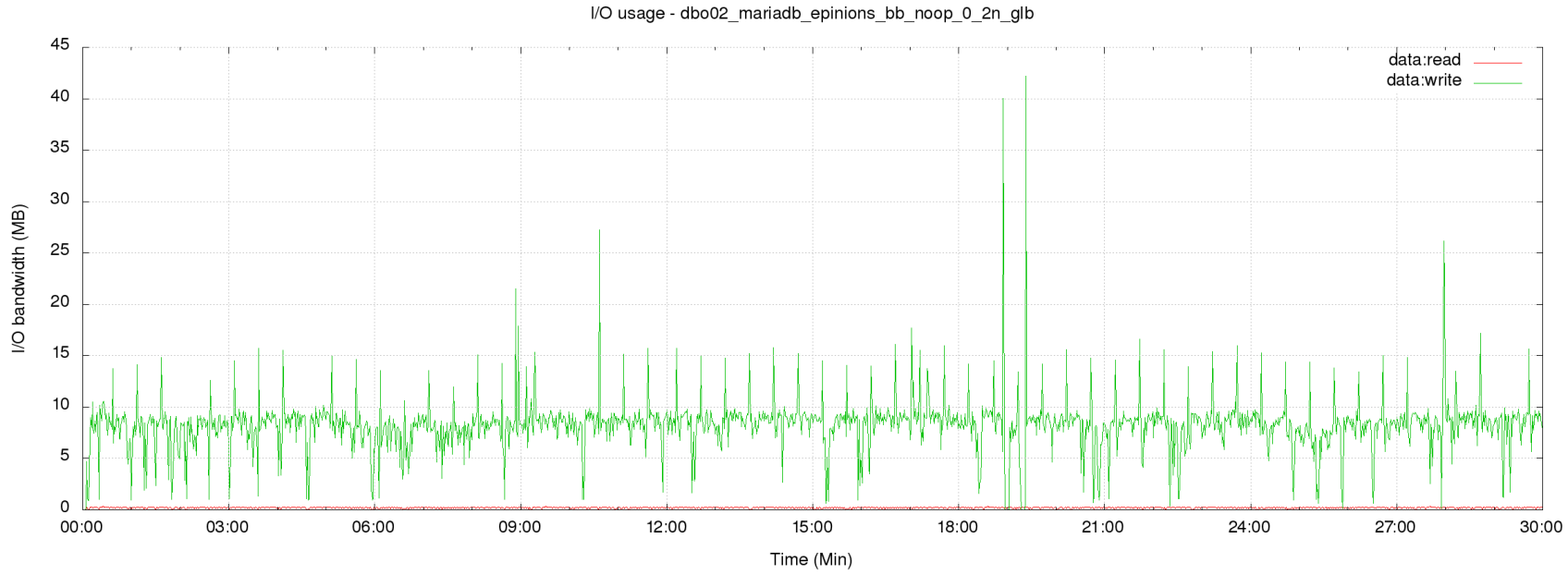
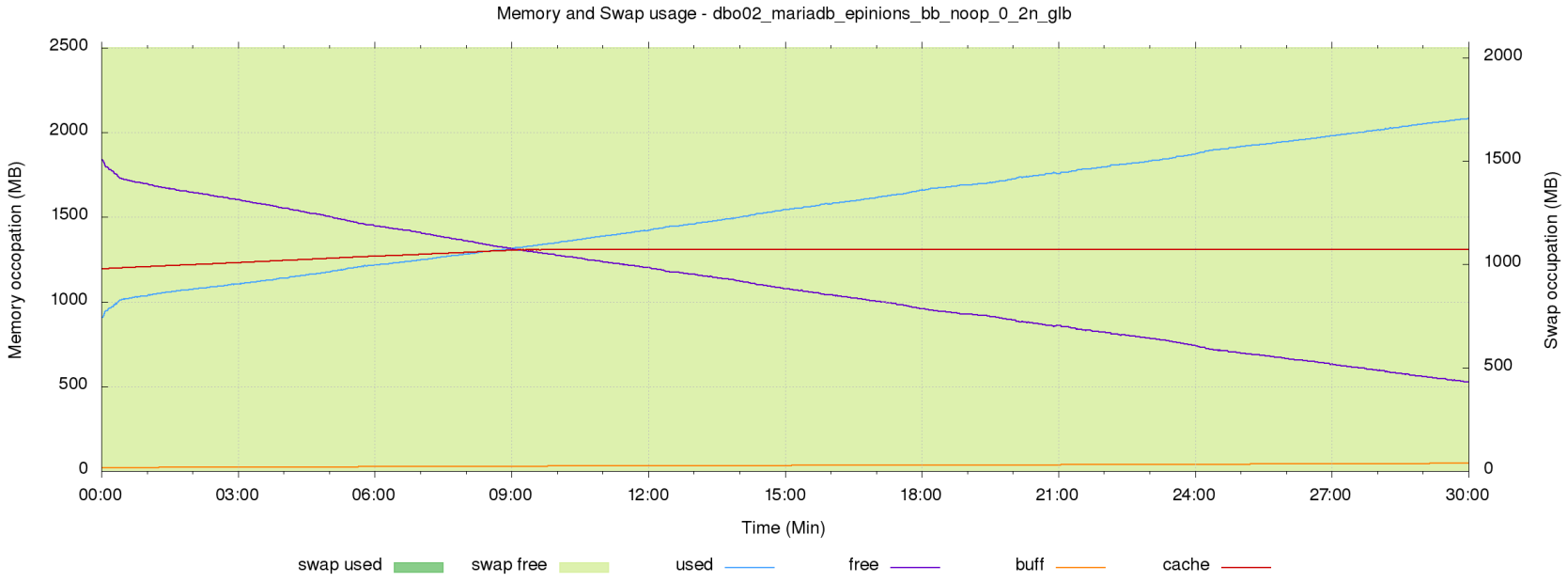


Figure 4.26: MariaDB Epinions benchmark: disk load of node two in streaming replication with GLB balancer

Figure 4.27: MariaDB Epinions benchmark: memory and swap load of node two in streaming replication with GLB balancer



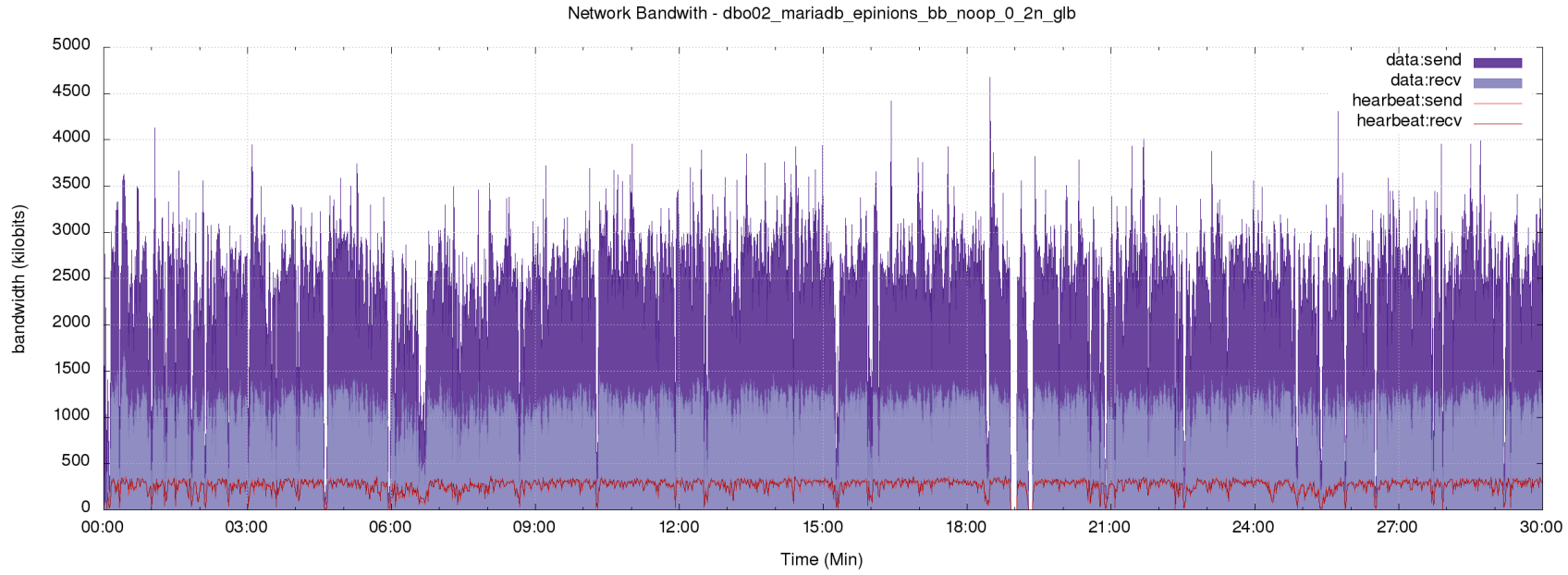


Figure 4.28: MariaDB Epinions benchmark: network throughput of node two in streaming replication with GLB balancer

4.3 PostgreSQL benchmark

PostgreSQL 9.3 DBMS has not the possibility to be clustered in a solution with multiple read-write nodes. To have a comparison with the DBMS MariaDB, the configuration that comes nearest is the replication mode: it consists in a read-write master node while the slave node is synchronized with the master in read-only mode. The only balancer available for PostgreSQL in clustered solution is *PgPool-II*. As mentioned in the paragraph 3.3.3, it has the capability to balance the read-only load between different nodes, and this feature is not available on others balancer such as *HAProxy* and *GLB*.
The number of tests run is 16 for each workload because the number of balancers utilized is just one. The test organization is like MariaDB so, at the beginning, the two nodes are joined to give the throughput of the balancers solution. Like the precedent DBMS, the tests with the balancer are divided in two parts: one test is a classic balancer where *PgPool-II* utilizes all nodes while the second test is run to determinate the overhead of the balancer: in this case *PgPool-II* is merely a pass-through from the benchmark to a single node.
The second part of tests are executed without the *PgPool-II*. Similarly to MariaDB two different tests are performed: one is on the base case, so the benchmark was directly connected to the DBMS without pass-through the balancer and only the master node is powered-on, while the second test is like the first with the slave node powered-on and in synchronized state. These tests are necessary to measure the overhead about the streaming replication solution.

As previously stated: the second purpose of this thesis is understand the impact of the I/O scheduler and the memory policy; given that, any test described below was run four times, and every time was changed one parameter of the machine. In particular we ran 4 different tests changing the following parameter: I/O scheduler (*CFQ* or *NOOP*) and memory policy (swappiness=0 or swappiness=60).

At the begin, the DBMS has been optimized to have the best performance.

4.3.1 DBMS configuration

The packages of PostgreSQL can be installed by *PostgreSQL Yum Repository*. To obtain a streaming replication solution like a production system several parameters and auxiliary scripts must be enabled. Generally the basic stuff and the tuning settings in the configuration file 4.2 are the same for any tests. These values were chosen after repeated trials and are a basic tuning to obtain high performances.
To achieve a test environment like a production one, we have to enable the *Write-ahead logging* (WAL). This feature is the journal of PostgreSQL and is mandatory when the

slave node joins to the master node. In fact, unlike MariaDB where when the join of the slave involves an automatic synchronization, PostgreSQL slave DBMS does a synchronization read to the old WAL log and if it is not present the only remaining option is to execute a master full backup on the slave, and redo a configuration of the slave. This operation is time-expansive and when the database is too large it is the last chance at disposal. Also in a high-availability solution where is possible to have a fail-over condition only for scheduled operations (like an software upgrade or maintenance activities) is impractical to execute a full backup whenever happens.
When the nodes have a synchronized state, the future changes are performed by the heartbeat network, and the WAL feature has a backup functionality to ensure the possibility of a Restore Point in Time (RPT).
Because the WAL files are numerous and the occupied space is too high, a dedicate file-systems has been created. Also, has been created two scripts to manage these file: *rsync_to_wal_archive.sh* and *rsync_from_wal_archive.sh*. We can invoke this script with *archive_command* and *restore_command* parameters: these operations are executed by the master node and by the slave node. The purpose of that is to copy the original file on dedicated file-systems; the slave DBMS invoke the restore command when has done the first join to the cluster or after a restore.
In a production environment the WAL files copied by the master are archived by the backup system on the TAPE (usually every hours) and the file-system used is very large but in our test environment the file system is small and a *crontab* job executes the deletion of this files.

Listing 4.2: PostgresSQL configuration's file

```
# Basic stuff
listen_addresses = '*'
max_connections = 110
lc_messages = 'en_US.UTF-8'
lc_monetary = 'en_US.UTF-8'
lc_numeric = 'en_US.UTF-8'
lc_time = 'en_US.UTF-8'
default_text_search_config = 'pg_catalog.english'
log_destination = 'stderr'
log_statement = none
log_min_error_statement = log
log_min_messages = log
client_min_messages = log
log_directory = 'pg_log'
log_filename = 'postgresql.log'
log_truncate_on_rotation = on
log_rotation_age = 1d
log_rotation_size = 0
log_line_prefix = '< %m >'
log_timezone = 'Europe/Rome'
timezone = 'Europe/Rome'
```

```
logging_collector = on
datestyle = 'iso, mdy'
constraint_exclusion = partition
default_statistics_target = 100
default_transaction_isolation = 'repeatable read'

# Tuning settings
maintenance_work_mem = 480MB
constraint_exclusion = on
checkpoint_completion_target = 0.9
effective_cache_size = 2816MB
work_mem = 18MB
wal_buffers = 32MB
checkpoint_segments = 64
shared_buffers = 960MB

# Wal settings
wal_level = hot_standby
wal_sync_method = fdatasync
wal_buffers = -1
max_wal_senders = 3
wal_keep_segments = 1
archive_mode = on
archive_command = '/opt/postgres/rsync_to_wal_archive.sh %p %f
   dbo02.hearbeat'
checkpoint_segments = 30
checkpoint_timeout = 35min
checkpoint_completion_target = 0.8

# Streaming replication settings
synchronous_standby_names = 'slave,dr'
hot_standby = on
synchronous_commit = on
```

Listing 4.3: PostgresSQL recovery configuration's file

```
standby_mode = 'on'
primary_conninfo = 'host=192.169.1.1 port=5432 user=replicator
   password=thepassword application_name=slave'
trigger_file = '/tmp/trig_f_postgres'
recovery_target_timeline = 'latest'
restore_command = '/opt/postgres/rsync_from_wal_archive.sh %p %f
   dbo02.hearbeat'
```

As you can see from the configuration file 4.2 the parameter *synchronous_commit* is set to *ON*. This parameter can change the behavior of the replication mode. If it is *ON* the DBMS's behavior fulfills a synchronized replica, while if it is *local* the DBMS is set in asynchronous mode. The configuration file differs between the master and the slave just on this parameter. In fact the value of *synchronous_commit* parameter on the slave

node is *local*, because if the master node is dead and the slave becomes the new master and if *synchronous_commit* is *ON* any operation will be stuck-ed. These parameters can be changed on the fly without the restart of the DBMS.

4.3.2 TPC-C workload results

The TPC-C workload utilized with the PostgreSQL DBMS has the same characteristics of the tests done with the MariaDB.
On the table 4.4 is reported the throughput results of the tests and there are some interesting results. As for the MariaDB, the first interesting results is that the best results are obtained with the *NOOP* scheduler I/O with the memory swap policy to the default value. The reason is yet the storage layer for the same reasons of the previous DBMS.
Another achievement is the unexpected throughput of the solution with the use of the balancer *PgPool-II*. In fact you may well notice that on average the values are cut by almost 64 percentage points compared to the non-balanced solution. This result indicates to us that this kind of solution goes against all the initial assumptions.

Table 4.4: PostgreSQL TPC-C results

TPC-C Workload				
mode	I/O noop		I/O cfq	
	swappiness=0	swappiness=60	swappiness=0	swappiness=60
n1	751.8255	**760.3116**	675.4058	644.3549
n1 + (n2)	643.1989	702.3574	594.7116	580.1802
n1 + n2 + PgPool-II	389.0048	**405.1093**	366.2339	365.5960
n1 + (n2) + PgPool-II	399.9360	412.2783	387.9132	350.7815

The Linux system where the PostgreSQL was configured in standalone mode has a very different behavior compared to the same tests performed with MariaDB. The first difference you may notice on the figure 4.29 where it is displayed the throughput and the latency of the DBMS. The average latency is about to 110ms and the trend of the throughput is not linear but have some negative behavior. These are caused by the DBMS's deadlocks that block some transactions until the first blocking transaction does not end. This new phenomenon has been tested by reading the DBMS log file.
As you can notice, the CPU load on the figure 4.29 confirms the deadlock phenomenon. In fact the CPU is not in IDLE state, while most of the CPU is in used state. This means also that the system had not a network or disk I/O bottleneck and the disk graph 4.31 confirms it. The write peak value is less than the maximal bandwidth of the disk, and occurs about every 30 seconds. The trend of readings is constant in time, and is certainly not a bottleneck. The network activity, that can be seen on the figure 4.33, is

the opposite of the MariaDB. The percentage of received data is higher than the sent data, while the heartbeat interface is used to transfer the WAL-log on the second node. Finally, the memory 4.32 has an higher value of cache, that justifies the low utilization of the I/O. Instead it is also possible to observe that the cache percentage has a slight increase, while the percentage of the used has a slight decrease. The swap is essentially free although there is a tendency to increase.

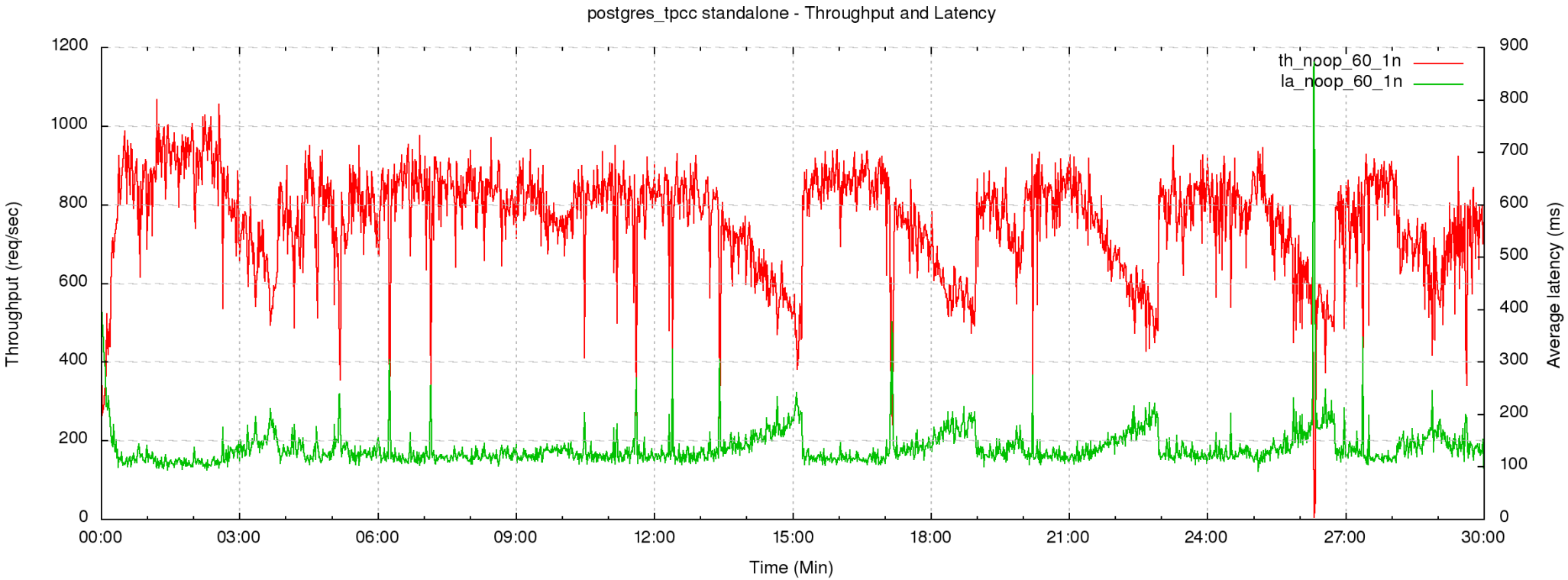
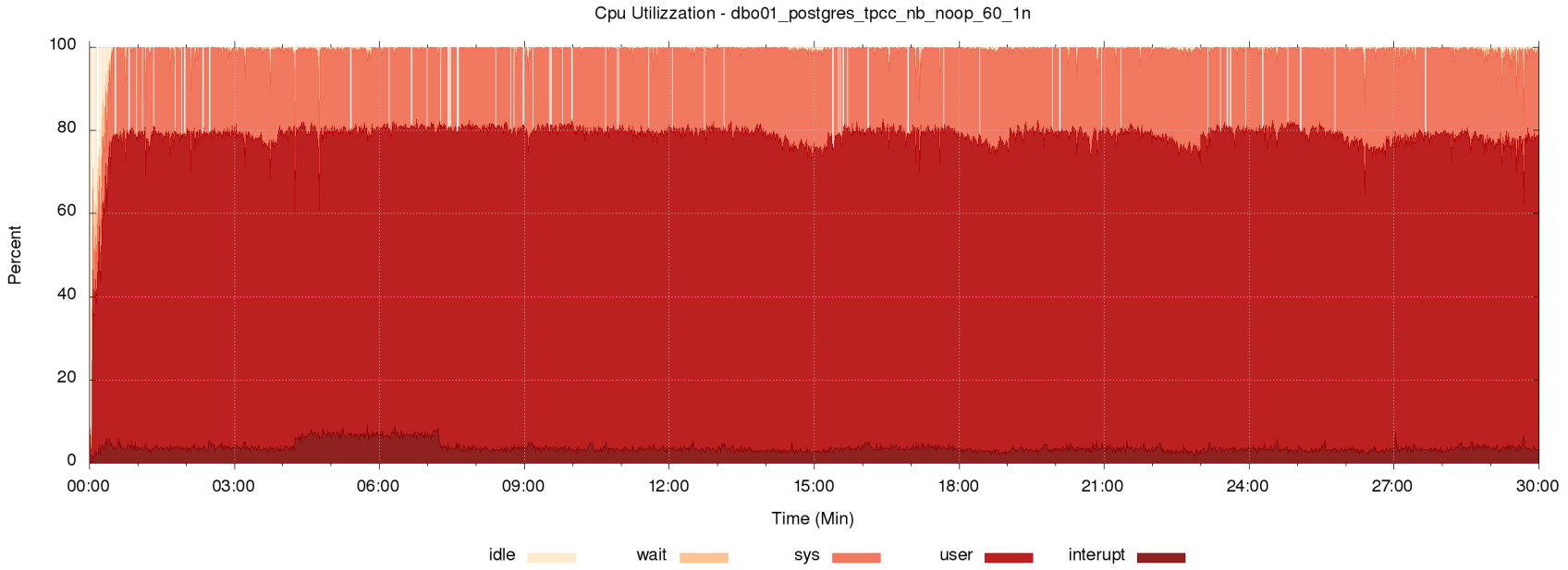


Figure 4.29: PostgreSQL Epinions throughput on a single node without a balancer

Figure 4.30: PostgreSQL Epinions benchmark: CPU load of node one in streaming replication with PgPool-II balancer



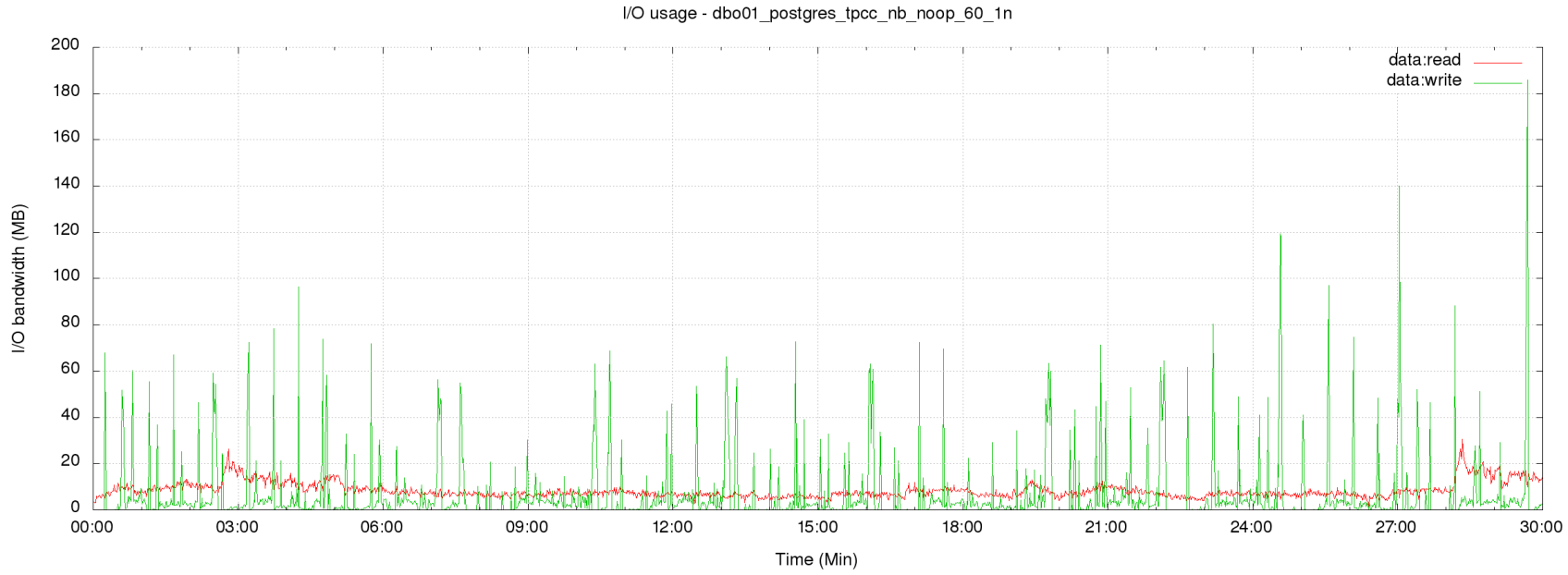
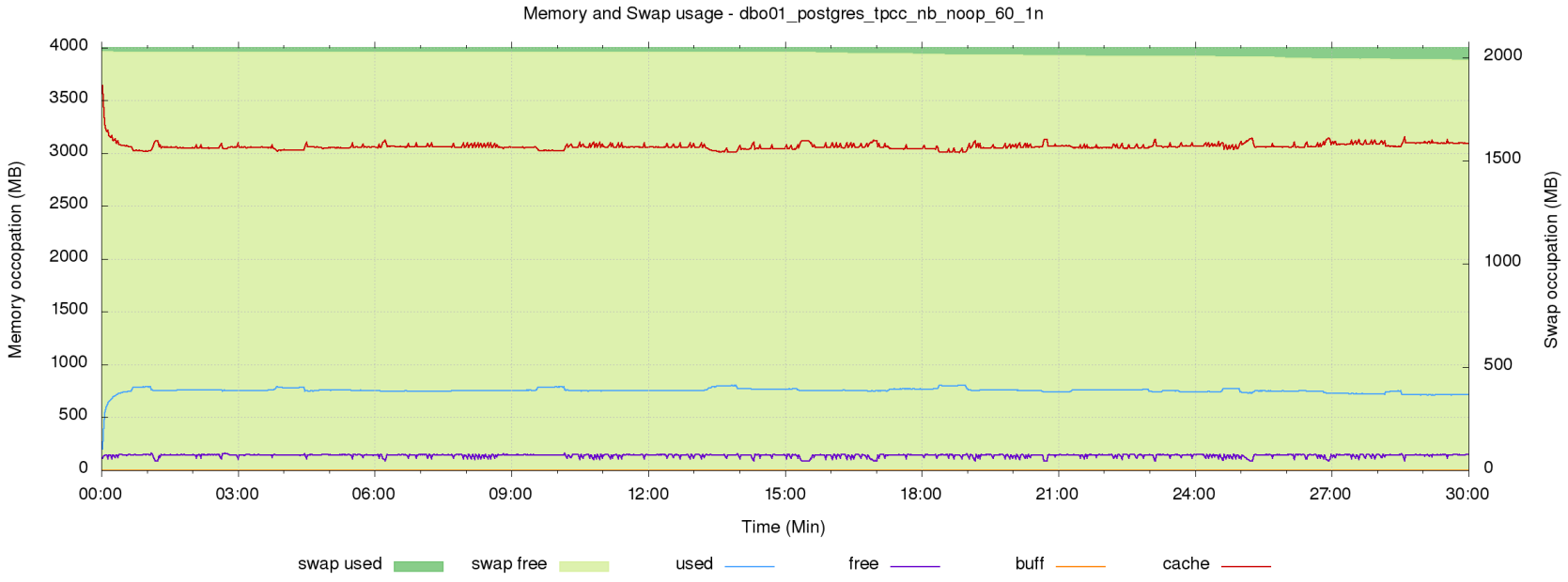


Figure 4.31: PostgreSQL Epinions benchmark: disk load of node one in streaming replication with PgPool-II balancer

Figure 4.32: PostgreSQL Epinions benchmark: memory load of node one in streaming replication with PgPool-II balancer



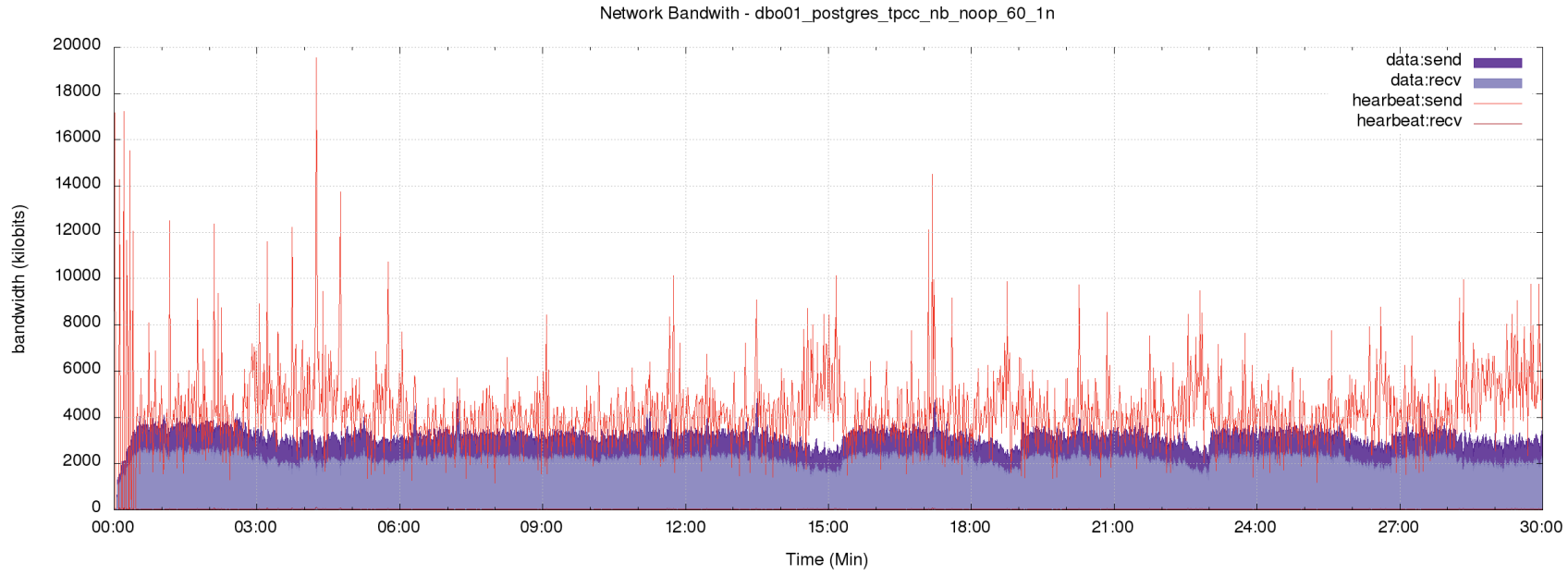


Figure 4.33: PostgreSQL Epinions benchmark: network throughput of node one in streaming replication with PgPool-II balancer

The workload TPC-C over the PostgreSQL in cluster mode has a very similar behavior of the solution in standalone, except for the fact that the throughput is 0.64% less. As we can see on figure 4.34 the latency has an average 260ms which is more than double of the previous, while the trend of throughput is not regular but almost sinusoidal.
Also in this case it is clear from the log of the DBMS that there are numerous deadlocks that causes this type of behavior. In addition, there is the presence of the balancer that is not conducive to the performance of the system. On the graphs 4.36 and 4.40 we can observe the behavior of the disk. From those we can notice that the node 2 has always activity of write, caused by the WAL log and by the DBMS's log; instead on the node 1 there are both read and write. We have to keep in mind that the the node 1 has the faculty of reading and writing while the node 2 is read-only.
The WAL log has sent traffic from the node 1 through the heartbeat network to node 2. The data's network is used mostly by node 1 while the node 2 use it scarcely.
The performance of the CPU, that we can seen on the figure 4.35 and 4.39, is the same of the standalone test. Idle state is not present on the node 1, while there is an high value of user state that means the DBMS is working. On the node 2 the system has an high percent of idle state. These trend justify the presence of deadlocks on the DBMS, as it is reported in the DBMS's log file.
The last interesting aspect of this system is the memory behavior. On the node 1, you may notice on the figure 4.37 that there is an high value of cache: it means that the database is cached on the memory and the percentage of write are the flush of memory. The used value shows a slight decrease, while the occupation of the swap is on the rise. On the second node it is possible to observe the behavior of the synchronization with the first node. In fact every 2 minutes the system drops the old page from the memory and reload the new one. This phenomenon causes the trend on the figure 4.41. The swap on the second node is never used.

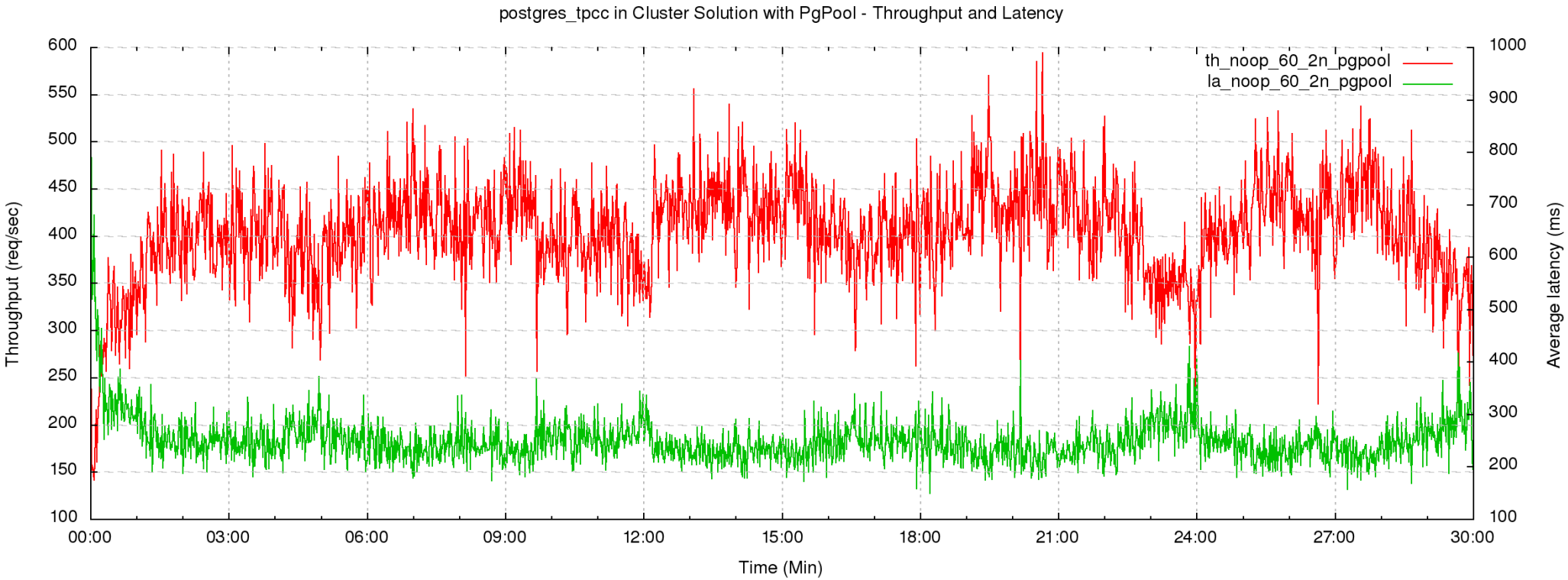
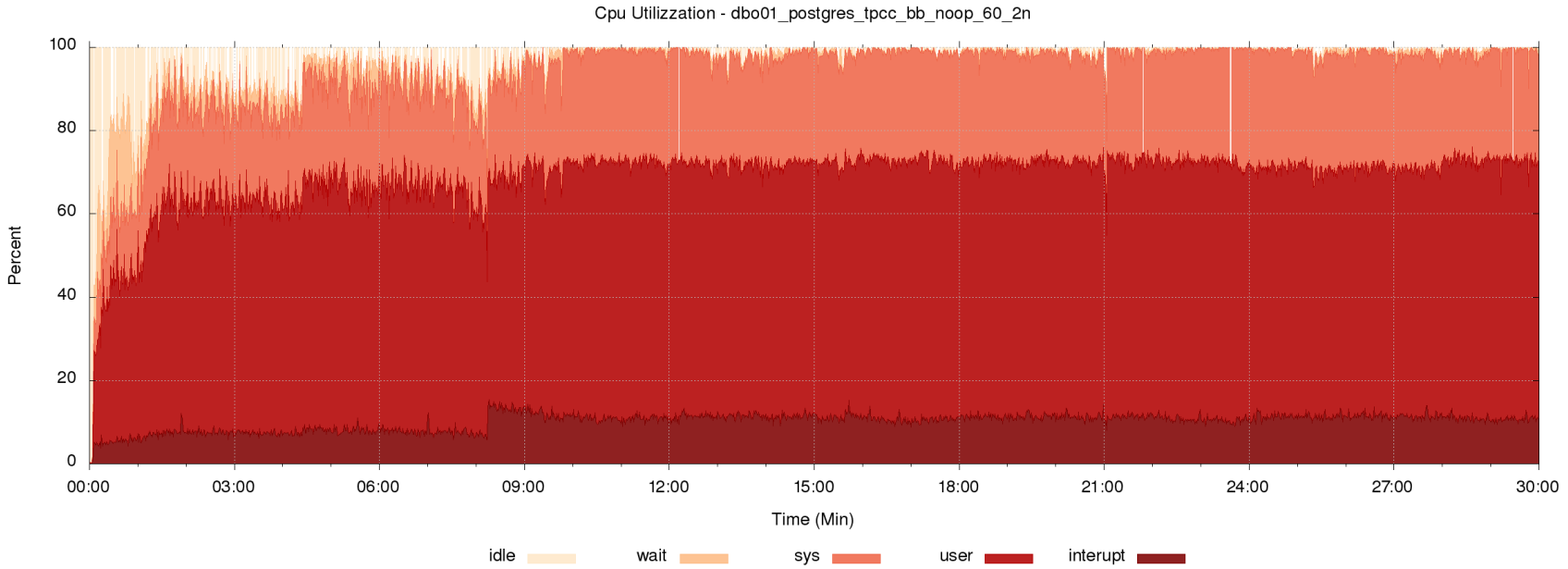


Figure 4.34: Epinions throughput on a cluster solution with two nodes active and the balancer PgPool-II

Figure 4.35: PostgreSQL TPC-C benchmark: cpu load of node one in streaming replication with PgPool balancer



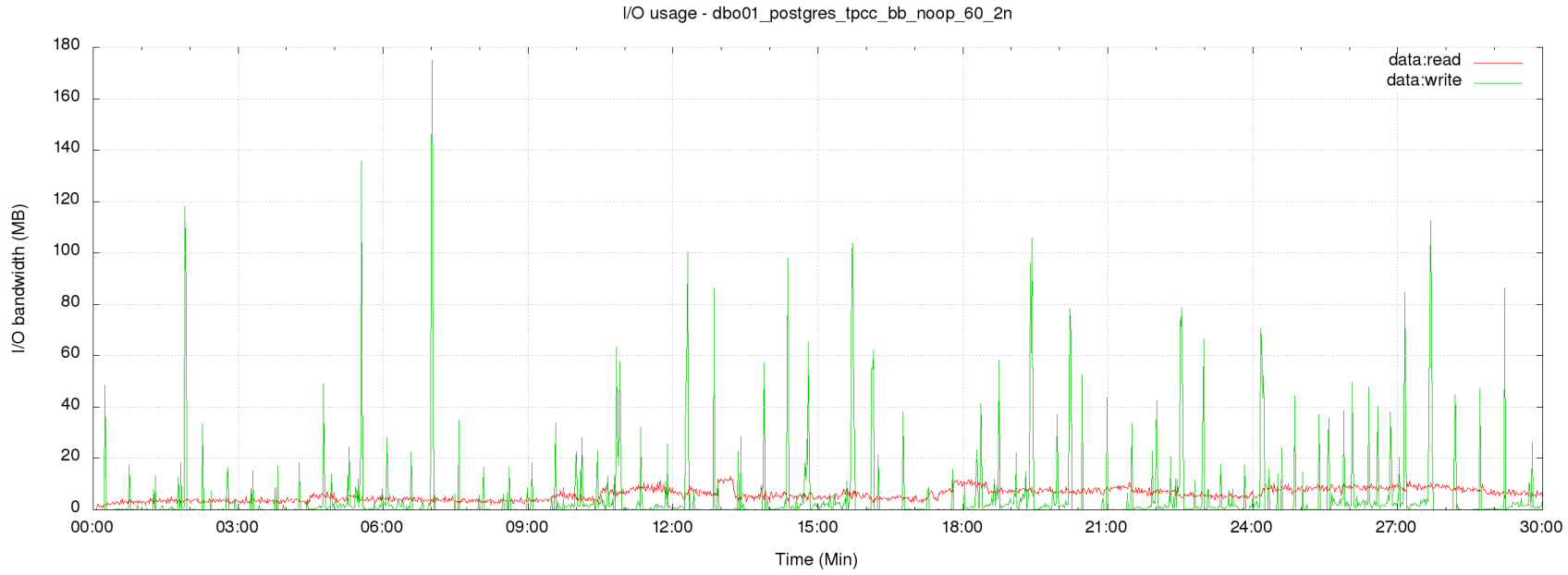
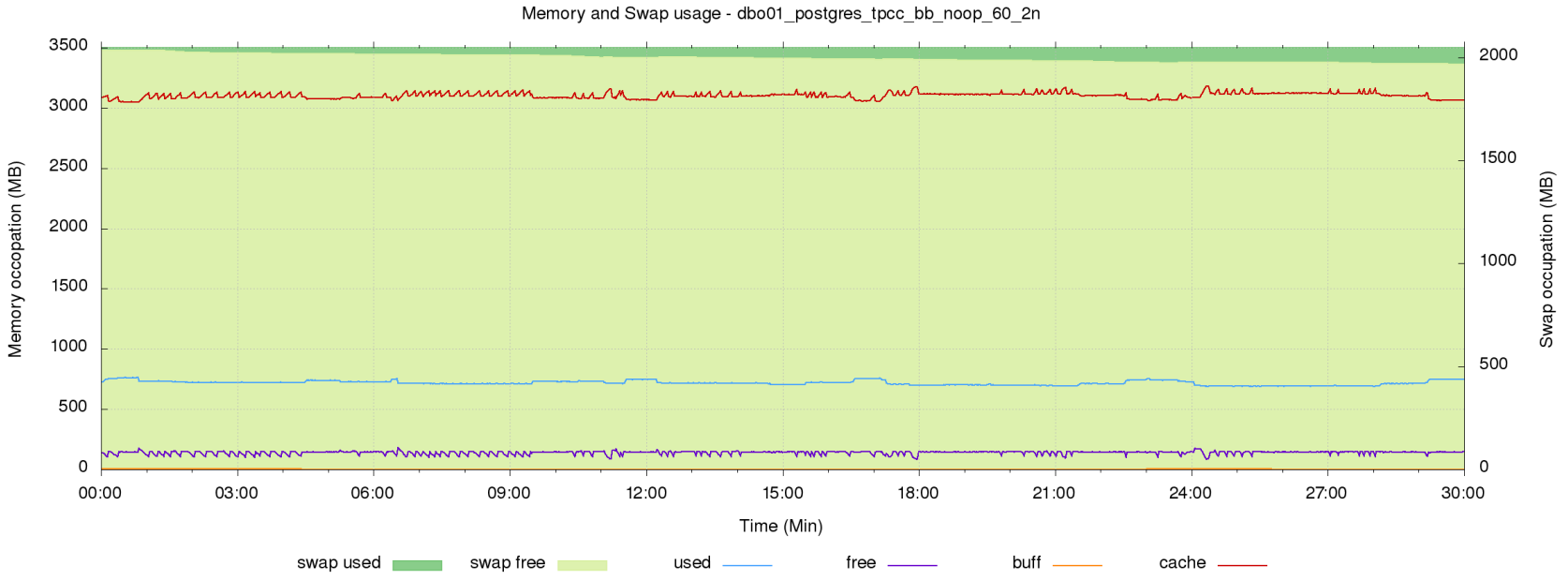


Figure 4.36: PostgreSQL TPC-C benchmark: disk load of node one in streaming replication with PgPool-II balancer

Figure 4.37: PostgreSQL TPC-C benchmark: memory and swap load of node one in streaming replication with PgPool balancer



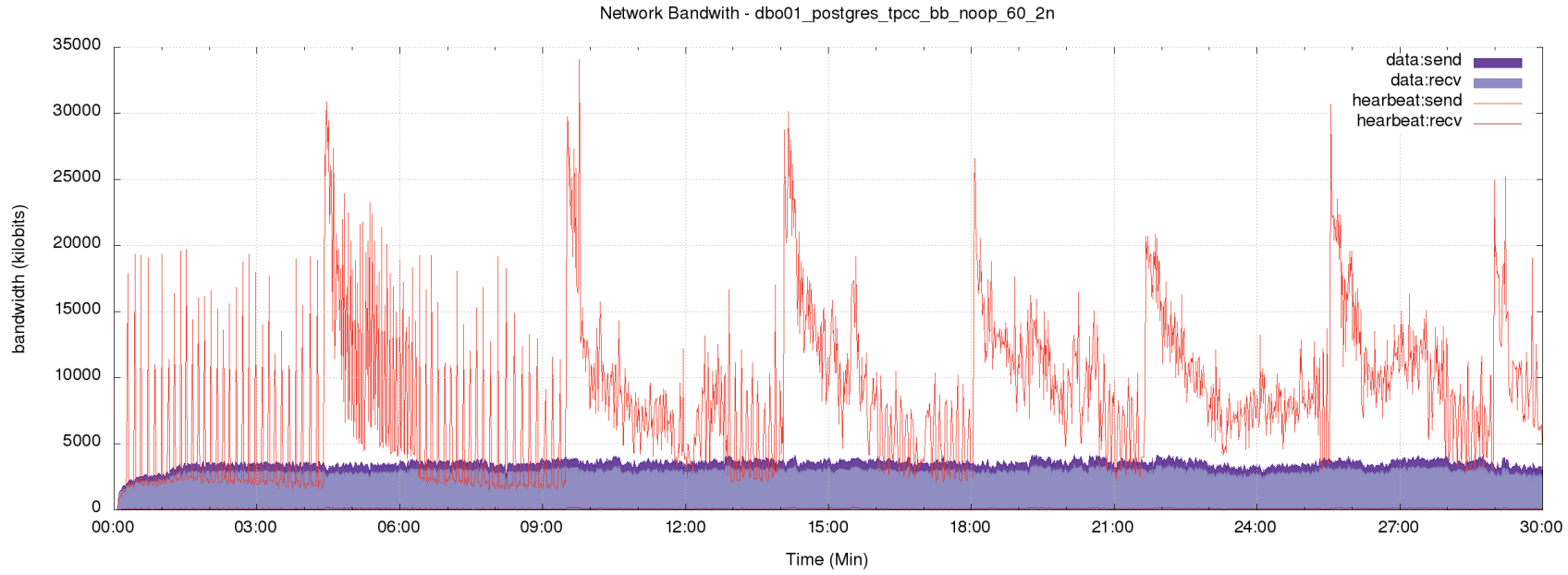
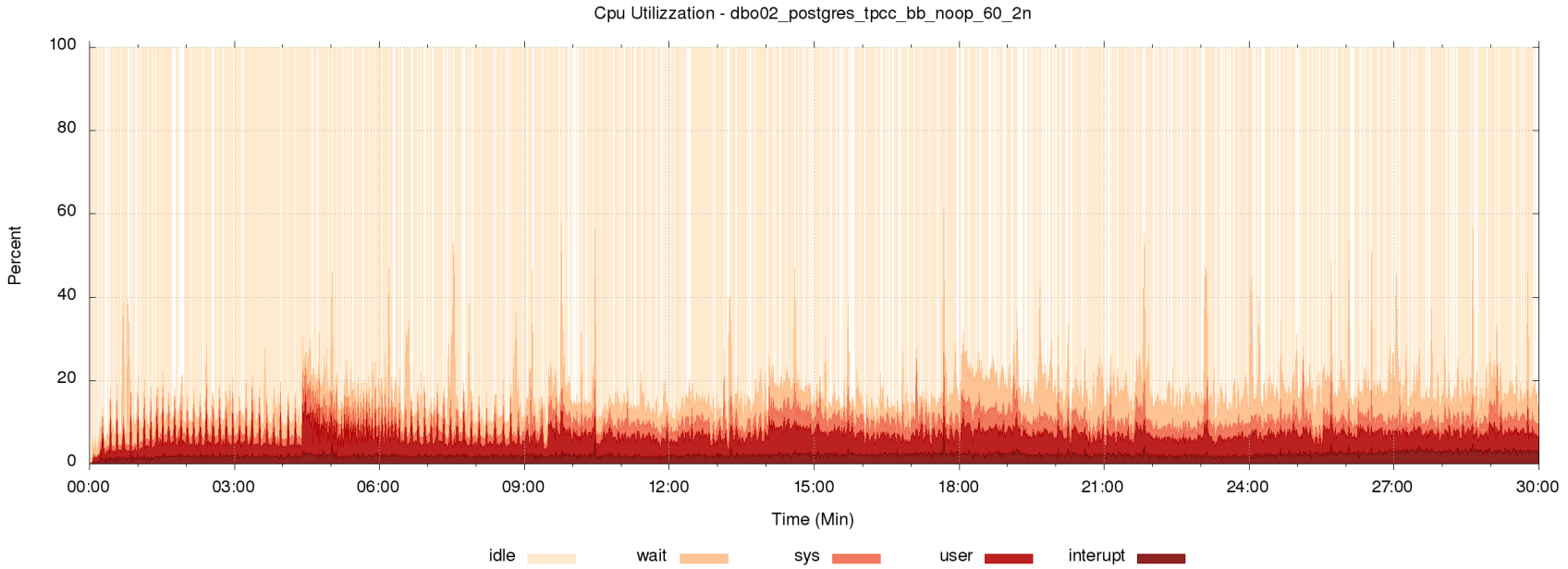


Figure 4.38: PostgreSQL TPC-C benchmark: network throughput of node one in streaming replication with PgPool-II balancer

Figure 4.39: PostgreSQL TPC-C benchmark: cpu load of node two in streaming replication with PgPool-II balancer



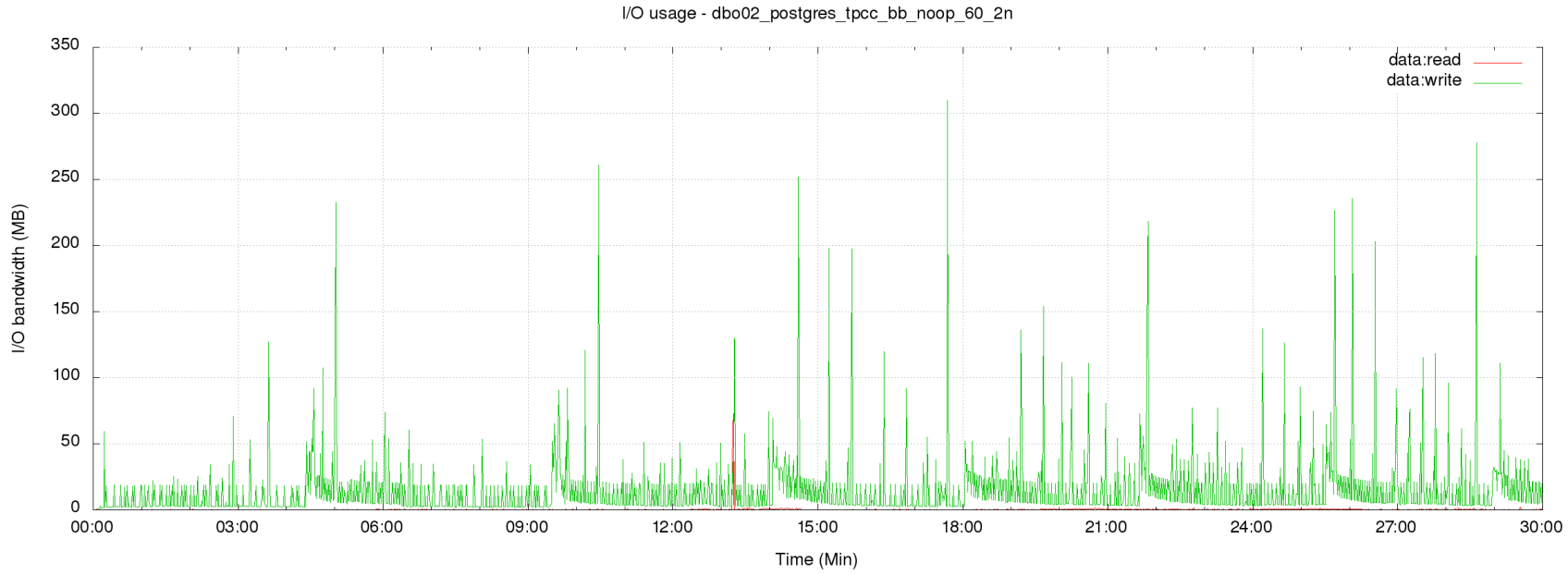
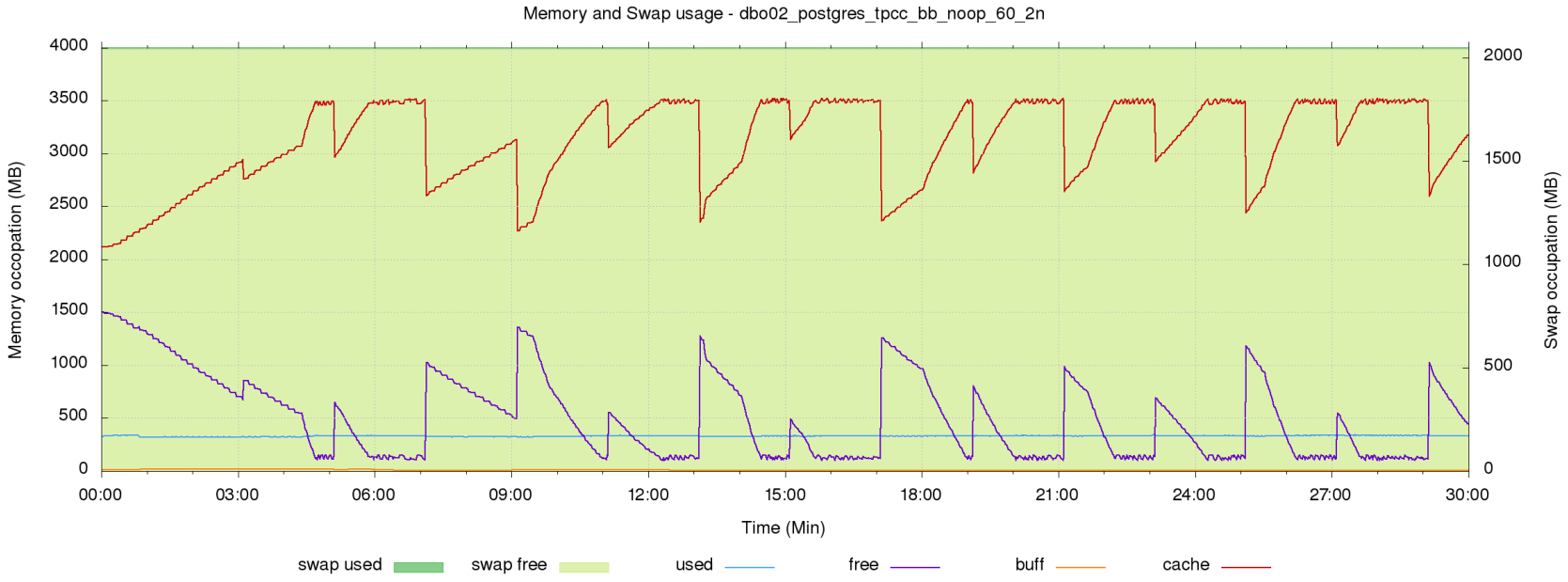


Figure 4.40: PostgreSQL TPC-C benchmark: disk load of node two in streaming replication with PgPool-II balancer

Figure 4.41: PostgreSQL TPC-C benchmark: memory and swap load of node two in streaming replication with PgPool balancer



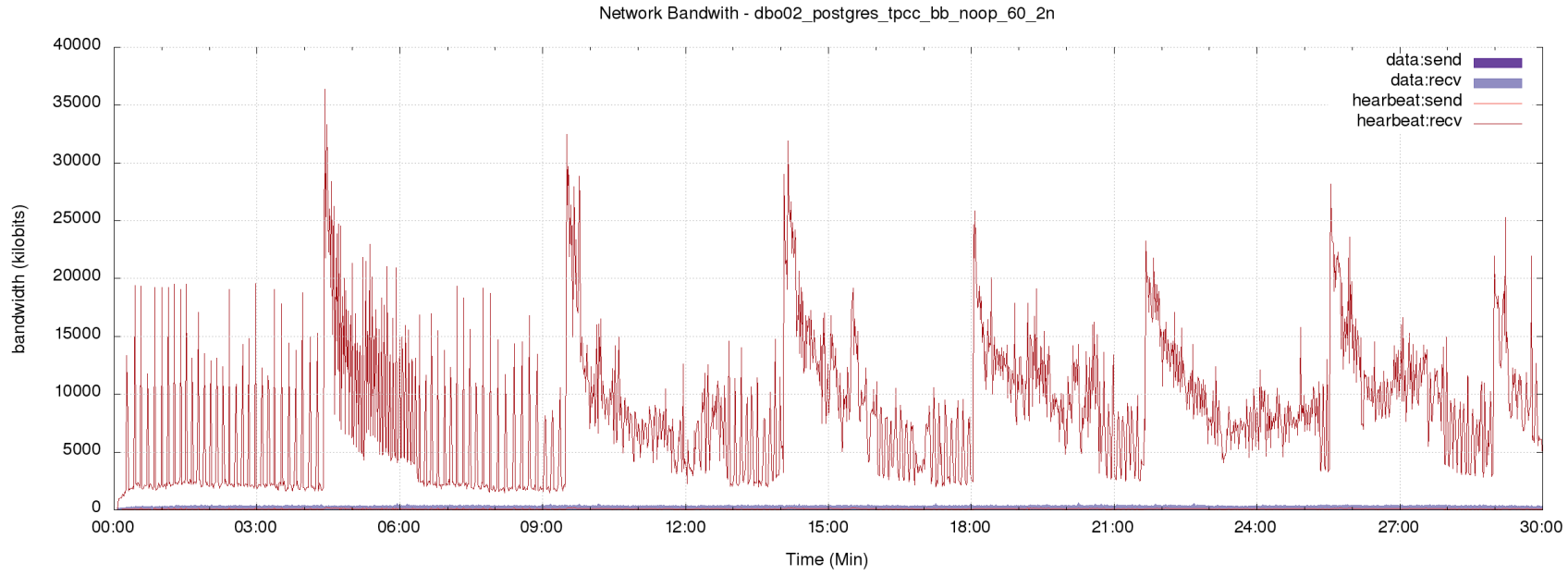


Figure 4.42: PostgreSQL TPC-C benchmark: network throughput of node two in streaming replication with PgPool-II balancer

4.3.3 Epinions workload results

The Epinions workload utilized for the PostgreSQL DBMS has produced two main results displayed on the table 4.5. The first one is that the two best configurations are with the scheduler I/O set to *NOOP* or *CFQ* and the swappiness set to 0. This is the same result of MariaDB and how stated in the previous paragraph the reason is due again to the type of storage layer used. This type of workload does not use the swap because the database dimension is less than the RAM of the hosts. This is the reason because with the swappiness setting to 0 there are better results.
The second main result that can be noted on the table is that the solution with a balancer is worse than without. In this case the balancer reduces about by 240% the throughput of the system. This is the second result that contradicts the initial expectations: the solution with the use of a balancer should have best performances.

Table 4.5: PostgreSQL-Epinions results

TPC-C Epinions				
mode	I/O noop		I/O cfq	
	swappiness=0	swappiness=60	swappiness=0	swappiness=60
n1	**9453.1875**	9332.3414	9387.3063	9271.4856
n1 + (n2)	9039.2923	8834.8602	8987.6464	8751.7635
n1 + n2 + PgPool-II	**2748.1216**	2563.9810	2567.4369	2422.7109
n1 + (n2) + PgPool-II	2506.8148	2452.5189	2362.8463	2300.1293

The system behavior with the Epinions workload has some trend like the TPC-C workload. The figure 4.43 displays the latency and the throughput of this workload, and from that it is possible to infer two important results. The first one is that the average latency is about 5ms which is the best result obtained value while the second one is that the trend of the throughput is very irregular.
From the figure 4.44 we can notice that the system has not a idle state, and the high percentage of the CPU is in used state to elaborate the tasked of the DBMS. As in previous cases, this behavior is not considered abnormal.
The behavior of network 4.47 shows that the heartbeat network sends every 1.30 minute the WAL log to the second node, while the percentage of send and receive from the data interfaces is the same of the TPC-C workload. The Disk load on the figure 4.45 shows that the storage layer is never used to read the database, because it is in the memory, while there is some peak of write that corresponds to the memory flush.
The reason is that the database was loaded into RAM, as we can observe in the graph 4.46. There is an higher value of cache that increases over time, as for the percentage of used while the percentage of the free memory decreases over time.

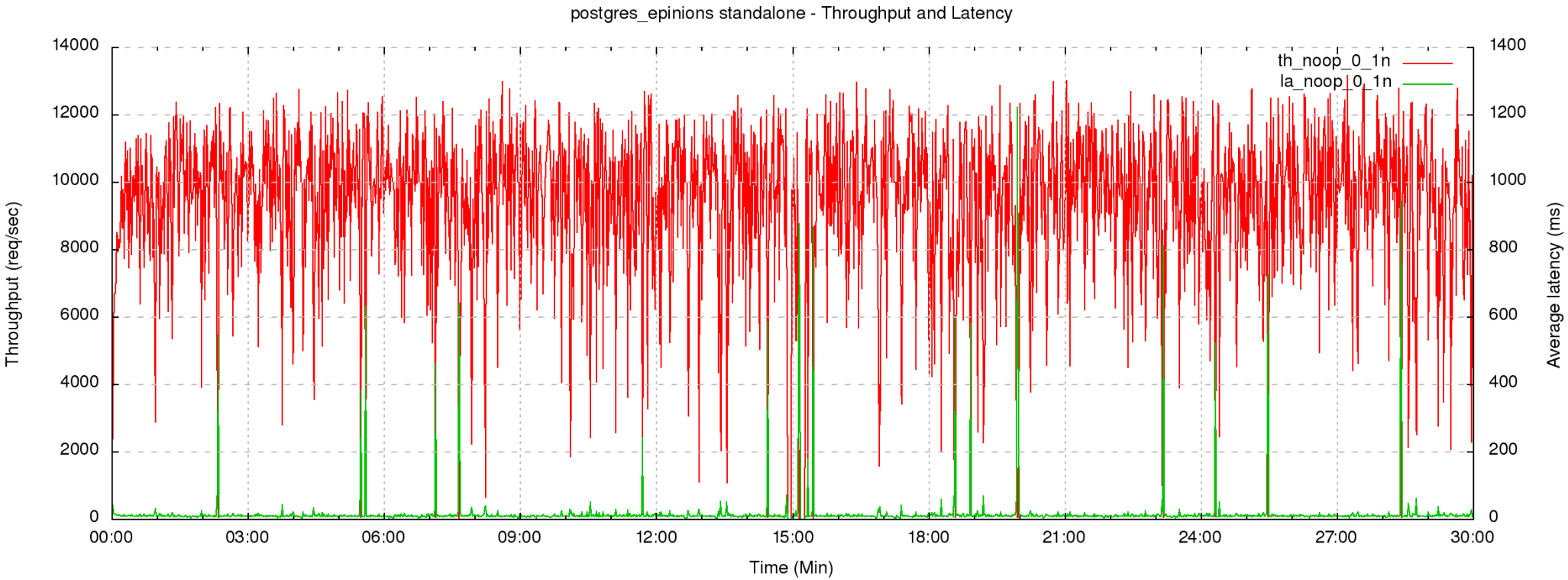
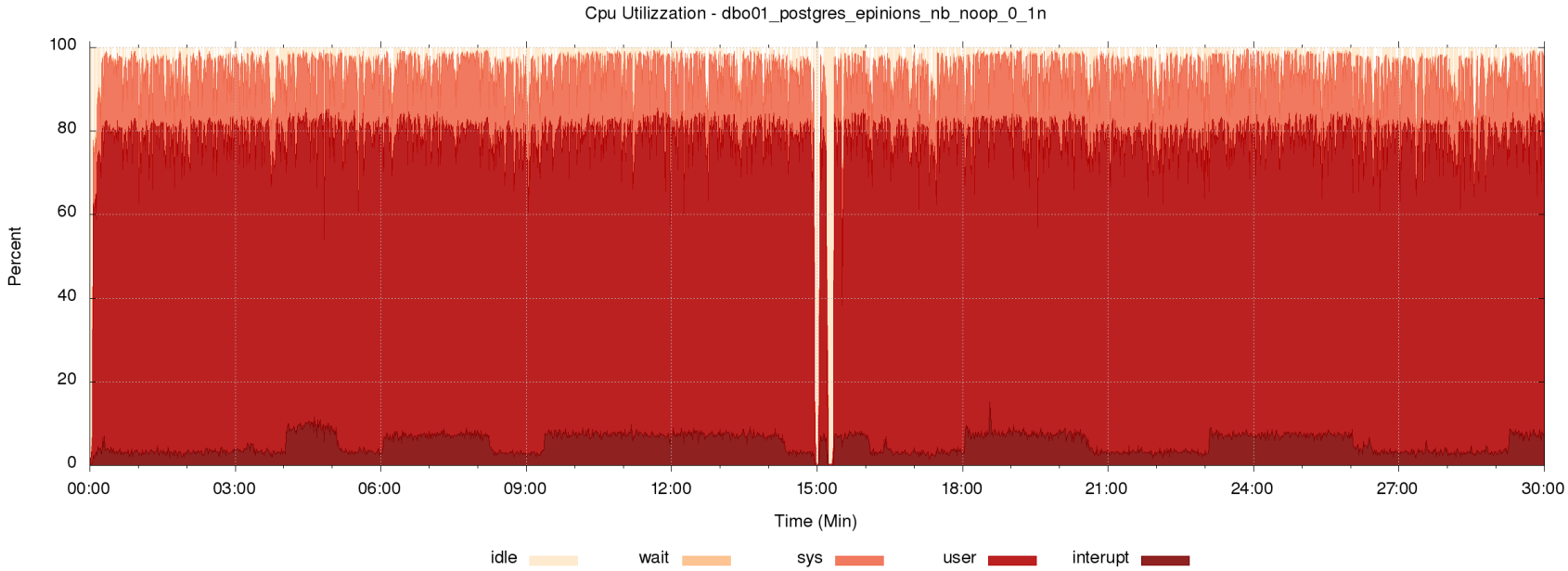


Figure 4.43: PostgreSQL Epinions throughput on a single node without a balancer

Figure 4.44: PostgreSQL Epinions benchmark: cpu load of node one in streaming replication with PgPool balancer



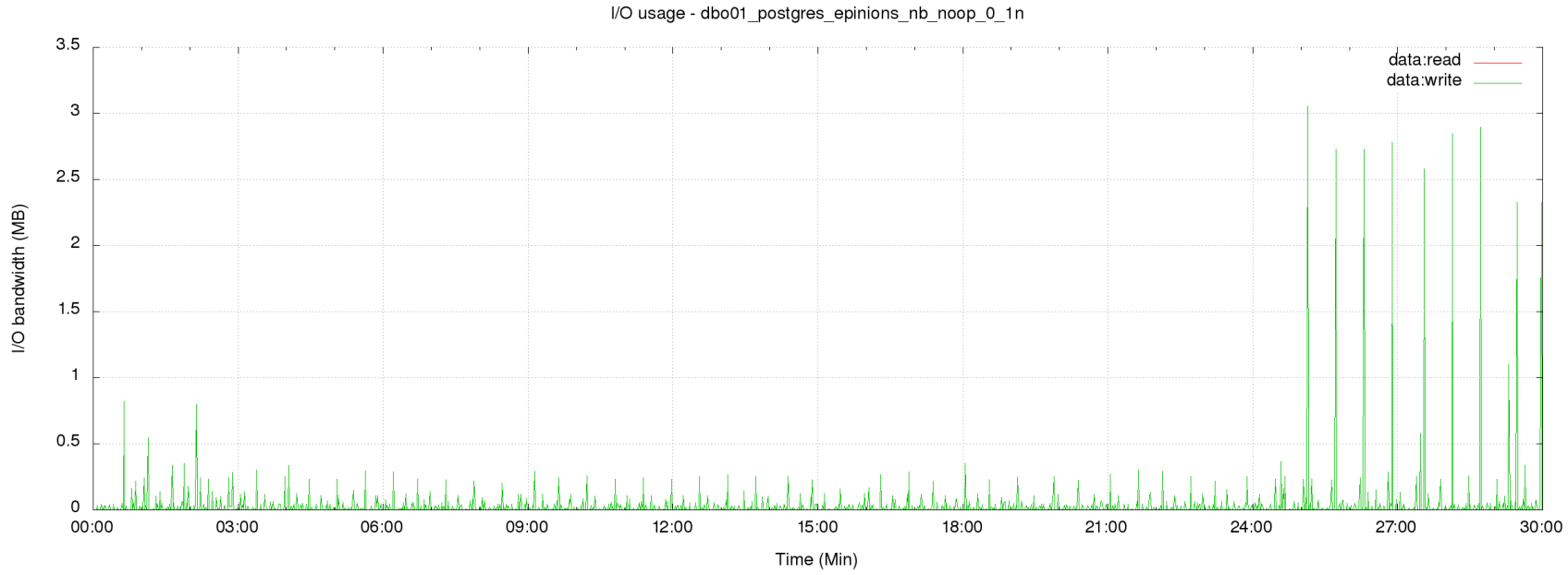
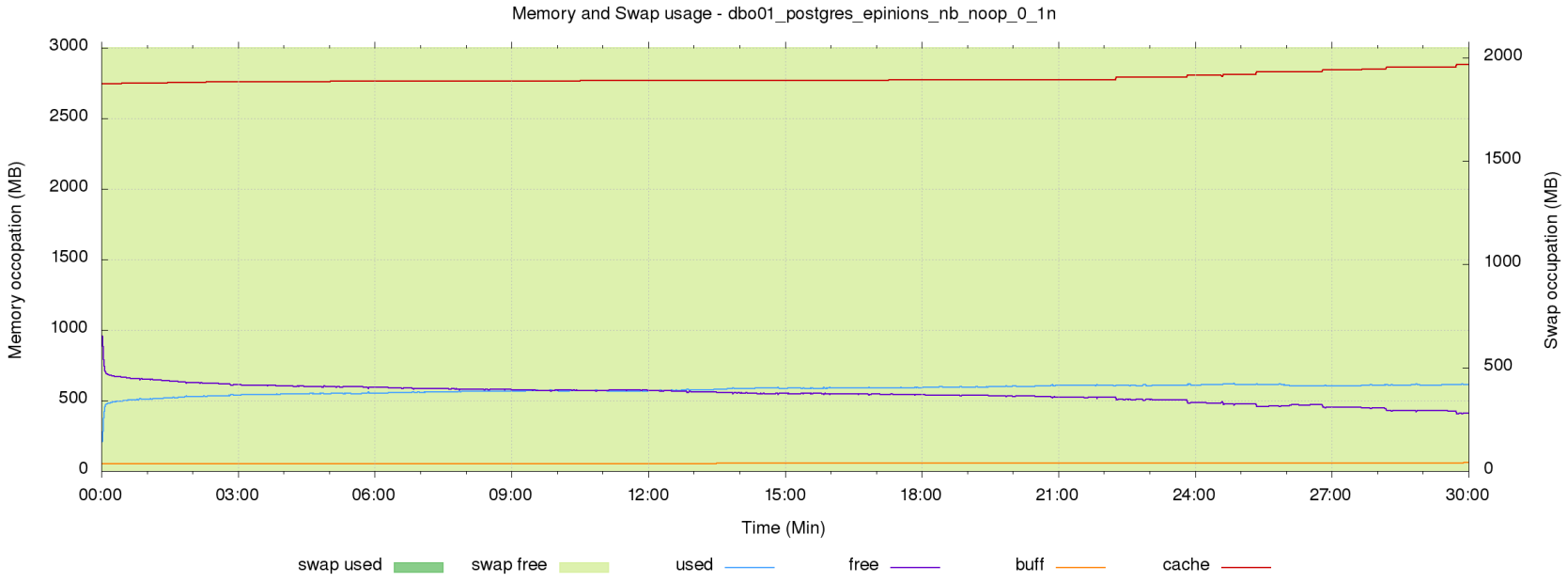


Figure 4.45: PostgreSQL Epinions benchmark: disk load of node one in streaming replication with PgPool balancer

Figure 4.46: PostgreSQL Epinions benchmark: memory load of node one in streaming replication with PgPool balancer



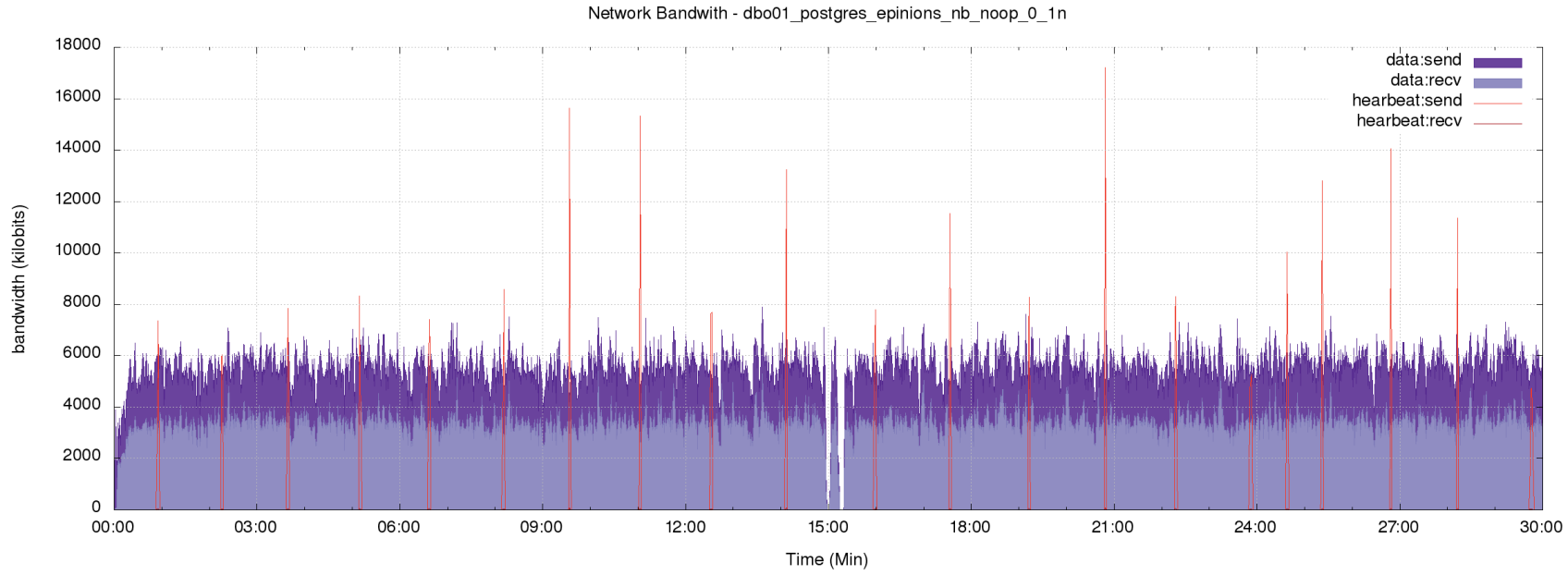


Figure 4.47: PostgreSQL Epinions benchmark: network throughput of node one in streaming replication with PgPool balancer

The overall behavior of the balanced solution can be seen in the graph 4.48. From those it is possible to notice that the system's latency is about of 50ms: that is a very lower lever than of the TPC-C workload. The throughput performance is rather similar to the test with the DBMS in standalone mode.
There are some considerations that reflect the behavior of the tests done in standalone mode and balanced test done with the TPC-C workload. The first of those is the trend of the CPU. As you may notice in the image 4.49 and 4.53, the node 1 has an higher percentage of idle state and the node 2 has not CPU activity. If you compare the test run in standalone we can note that the percentage of SYS state is the same, while the percentage of the user state is about half. There is also another interesting value: at 17th minute the node 1 has a dramatic 15% decrease of activity of the user, while the two node has a 5% increase to the same minute. This phenomenon leads to a reduction in throughput as you can see in the image 4.48.
Accordingly to the DBMS's tuning, the performance of the memory 4.51 4.55 is similar to the solution in standalone mode. In fact, the percentage of cache is very high, while the trend in the percentage used is slowly rising. The node 2 instead has a similar trend, although the cache values decrease to the same ones of the node 1. The memory used instead remains constant. The disk behavior, that can be seen on 4.50 4.54, has the same trend as a standalone solution. In fact, there is a total absence of read because the database is in RAM (cache), while writes are caused by the flush of changes to the database.
In conclusion, as you can see from the figures 4.52 and 4.56, the node 1 has a meager use of sent in data's network while we cannot note any use by node 2. The network heartbeat as always is used for transmission of the WAL log.

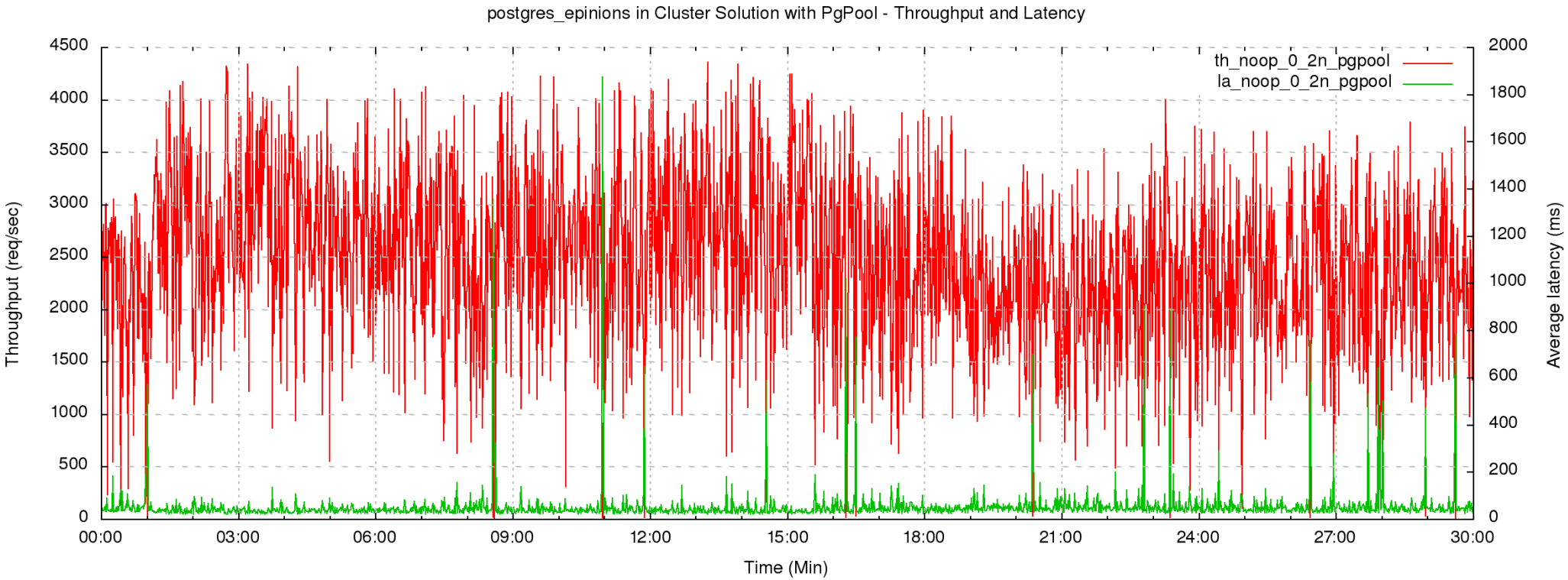
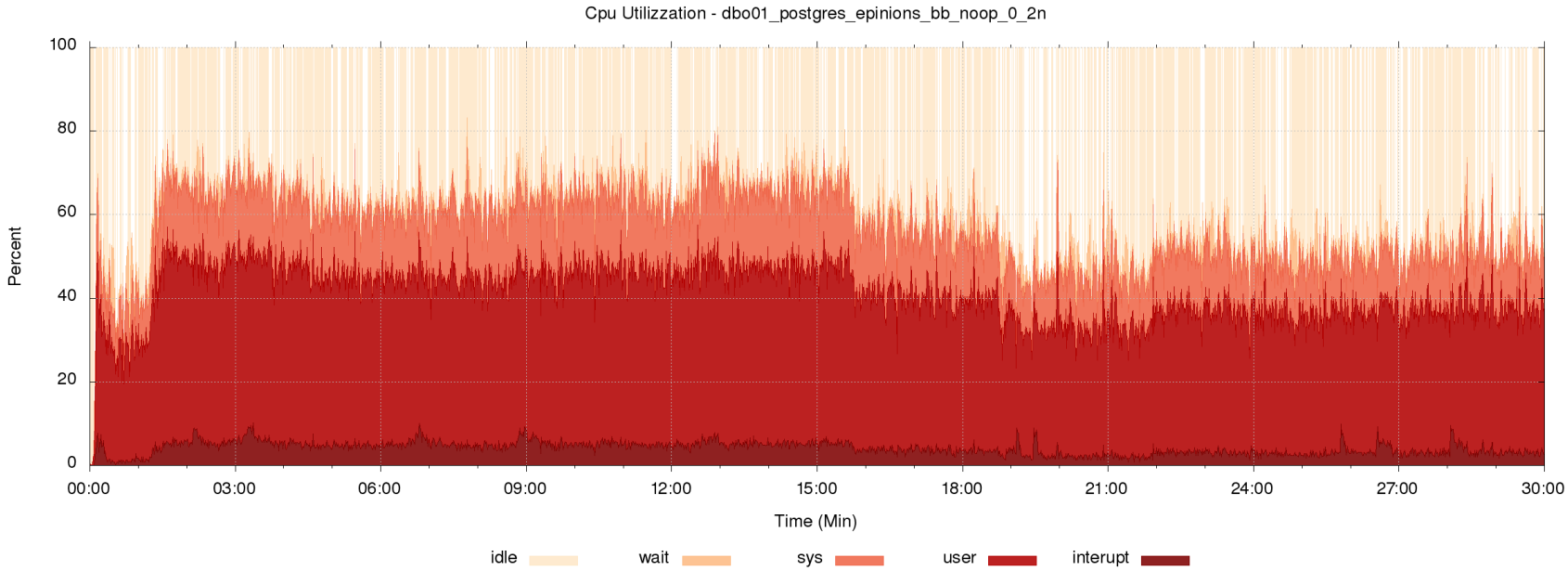


Figure 4.48: Epinions throughput on a cluster solution with two nodes active and the balancer PgPool

Figure 4.49: PostgreSQL Epinions benchmark: cpu load of node one in streaming replication with PgPool-II balancer



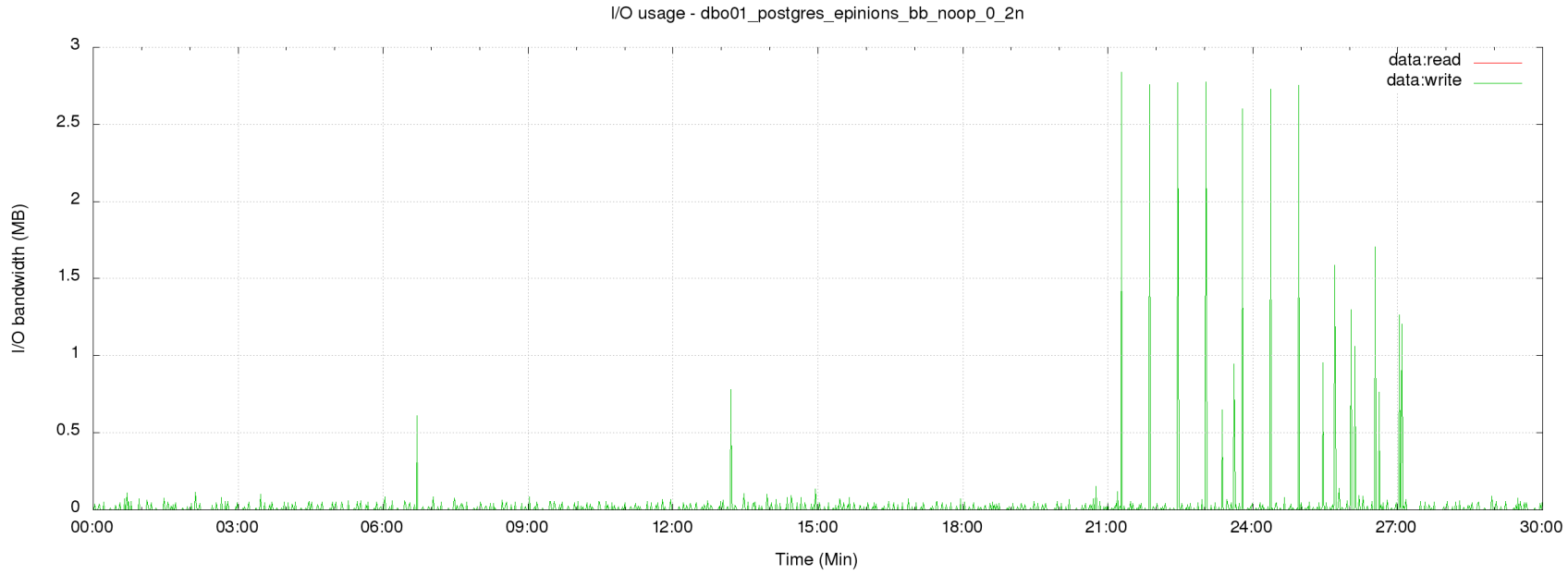
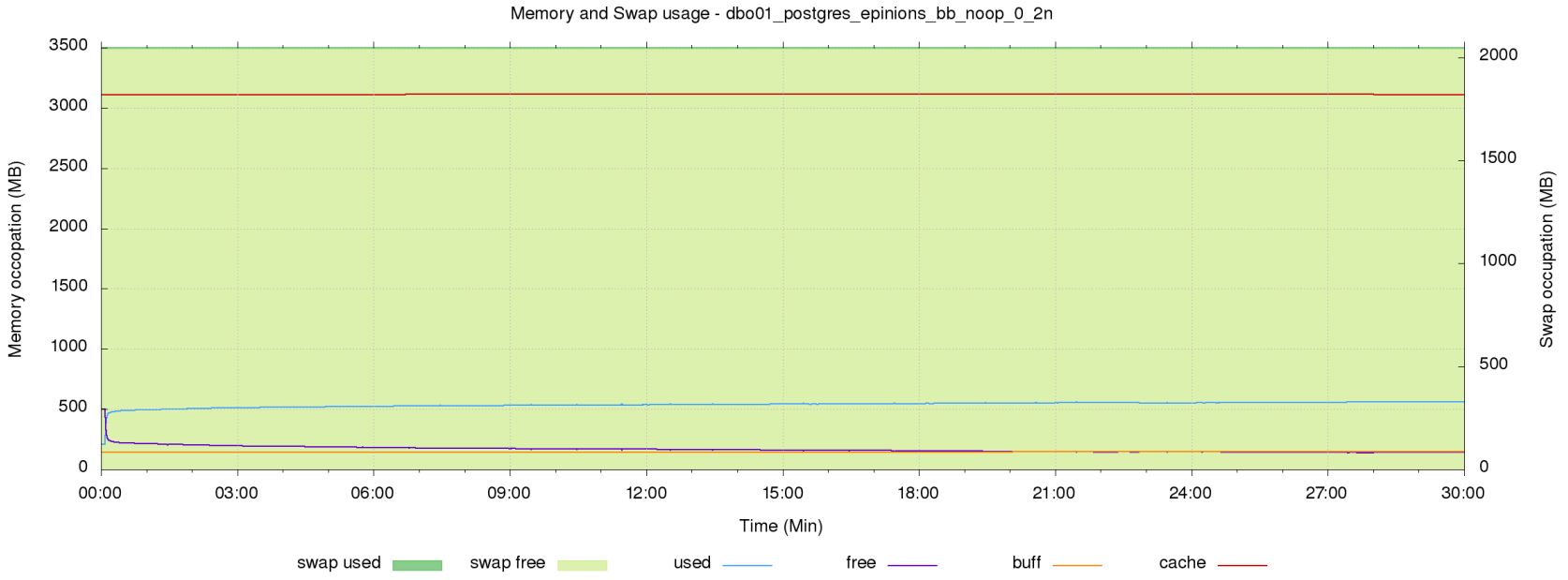


Figure 4.50: PostgreSQL Epinions benchmark: disk load of node one in streaming replication with PgPool-II balancer

Figure 4.51: PostgreSQL Epinions benchmark: memory and swap load of node one in streaming replication with PgPool balancer



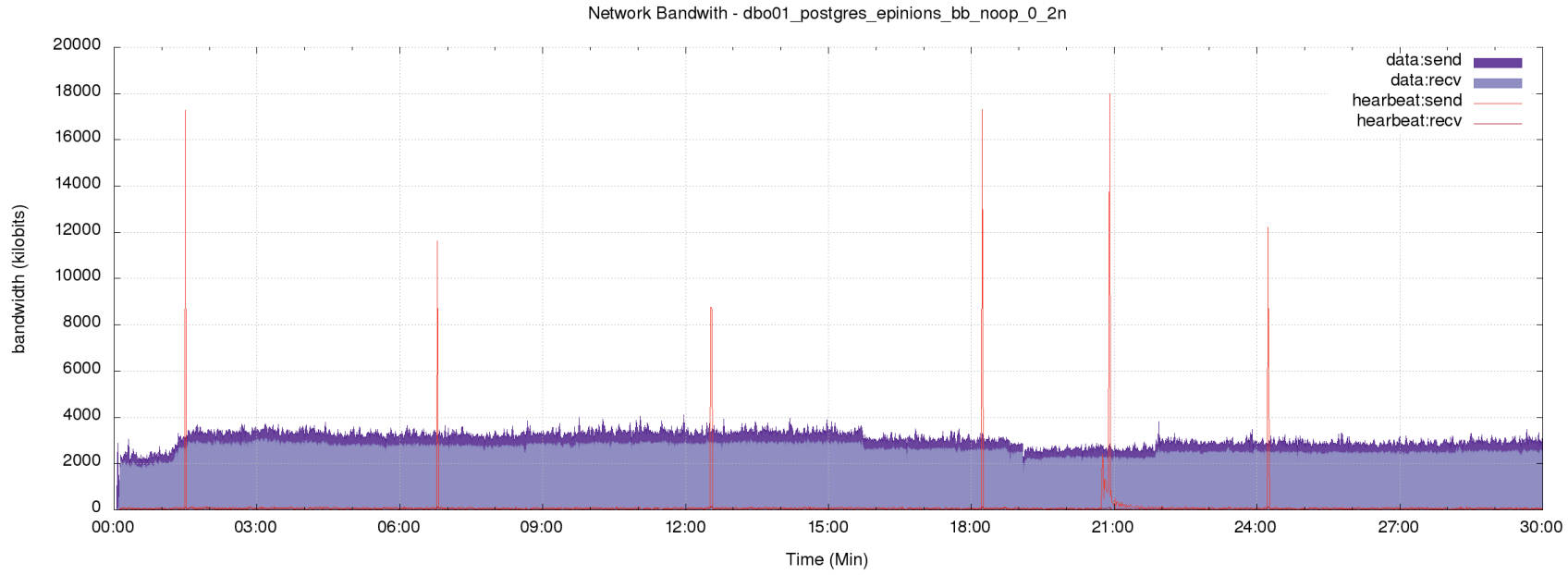
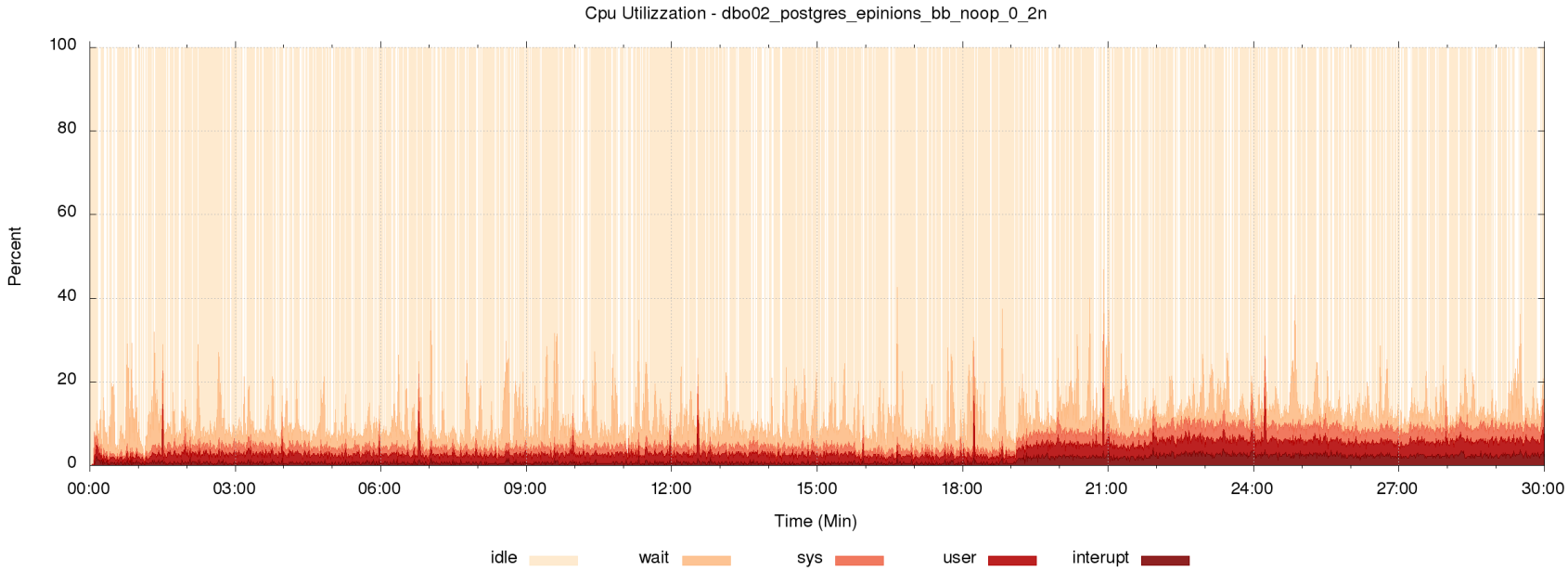


Figure 4.52: PostgreSQL Epinions benchmark: network throughput of node one in streaming replication with PgPool-II balancer

Figure 4.53: PostgreSQL Epinions benchmark: cpu load of node two in streaming replication with PgPoll balancer



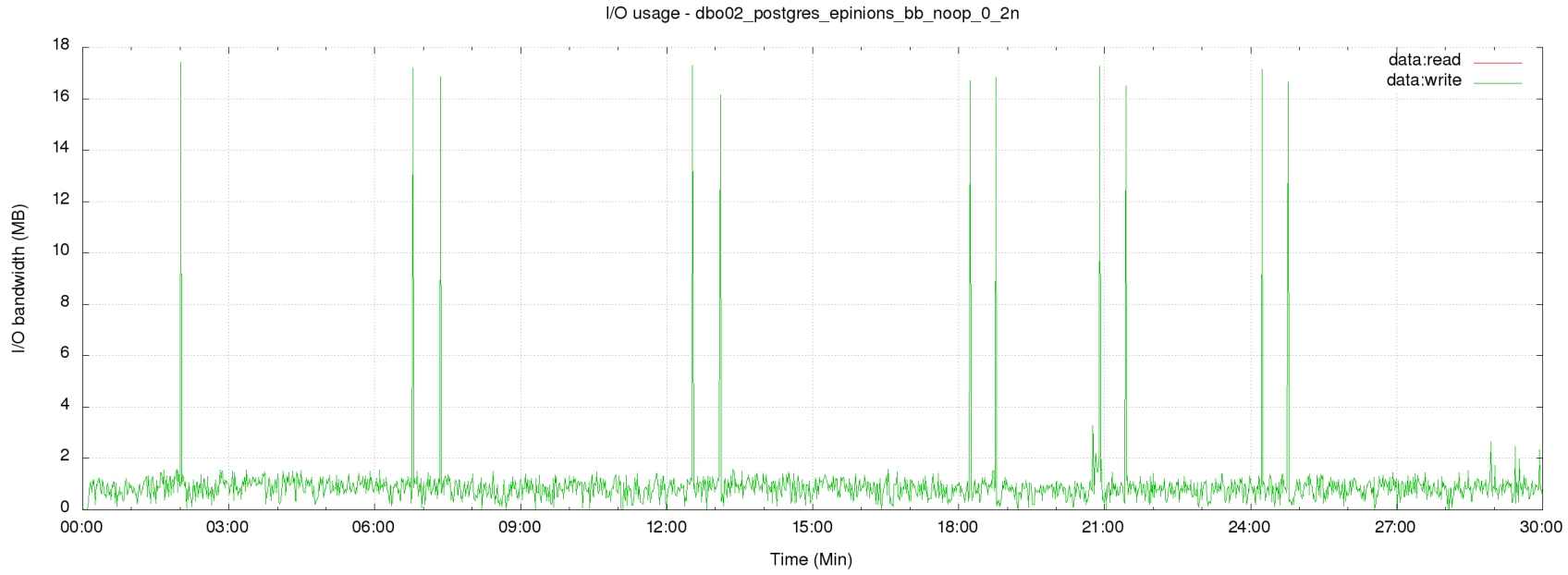
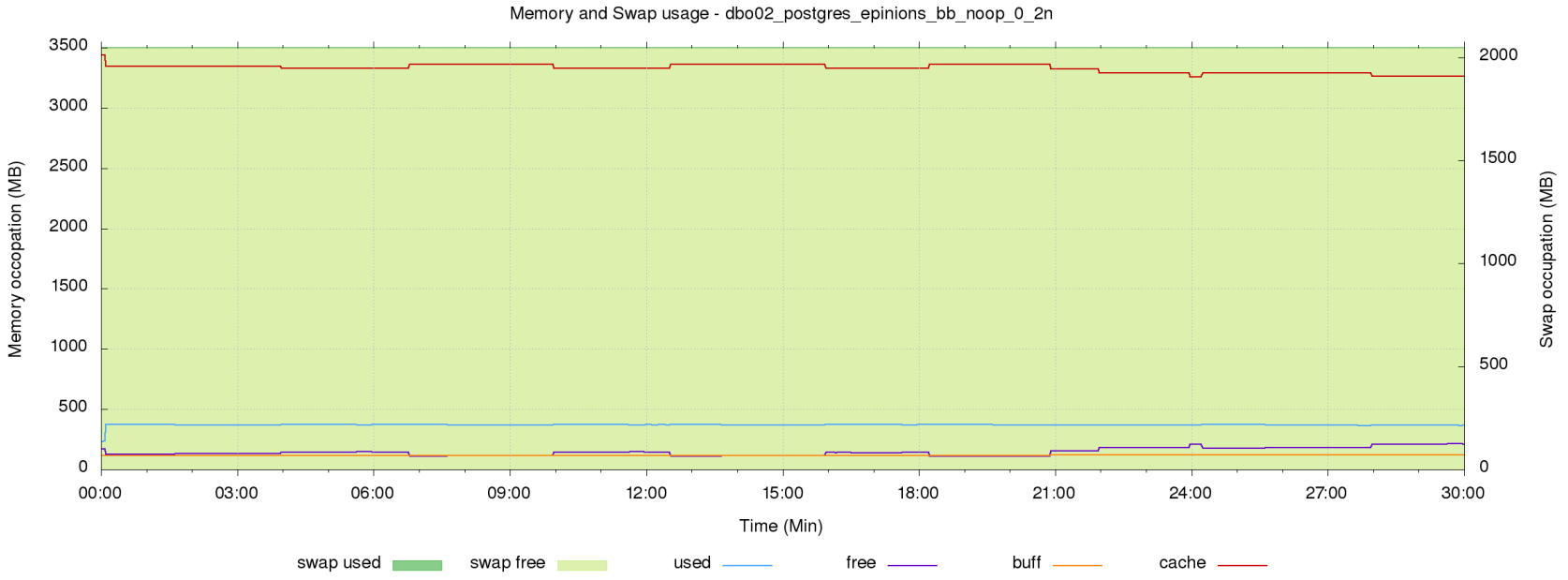


Figure 4.54: PostgreSQL Epinions benchmark: disk load of node two in streaming replication with PgPool balancer

Figure 4.55: PostgreSQL Epinions benchmark: memory and swap load of node two in streaming replication with PgPool balancer



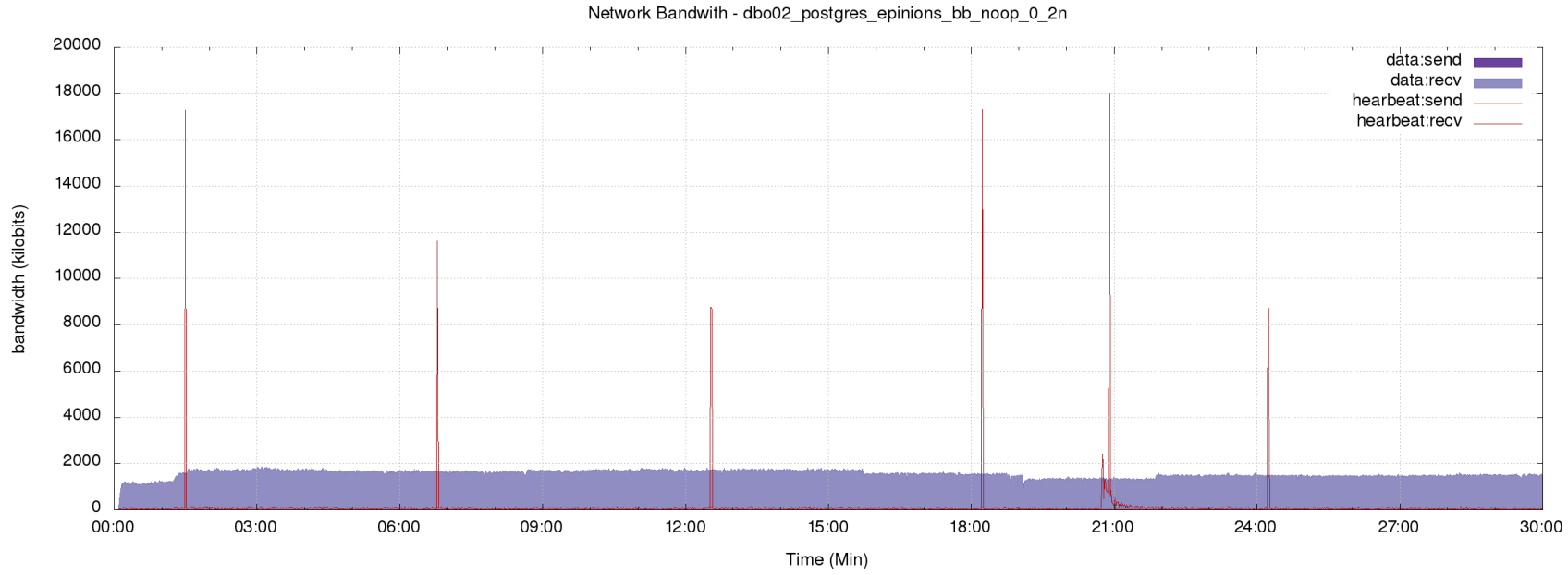


Figure 4.56: PostgreSQL Epinions benchmark: network throughput of node two in streaming replication with PgPool balancer

5 Conclusion

In this thesis, we have studied the major Open Source Database Management Systems that have synchronous replication features. We have studied their characteristics and which benefits give in terms of high-availability.
At the beginning of this paper we have covered the fundamental theory concepts about failure, error and fault, the background of the distributed system and the basic notions of the database. In chapter 1, we stated the characteristics that the DBMS must have to be a distributed database.
After that we have explained the basic concepts of a distributed DBMS and introduced the concepts of high-availability, because it is not implied that a distributed software is also reliable in terms of temporal continuity. Over the chapter 2, we presented the two main DBMS's category and their characteristics in terms of high-reliability.

At the end of the theoretical part, we have described the environment's topology used, particularly the characteristics of the hardware and software utilized. In chapter 3, we have analyzed the features of the two DBMS chosen in this essay and the characteristics of the balancer used to test the performance of the DBMSes. After that, we described the characteristics of the benchmark chosen, and the feature of the two workload used to test the performance. At the end of those chapters, we explained the importance of the scheduler I/O and the memory policy in every Linux operation systems.
After the theoretical background has been introduced, the chapter 4 explains the parameters used to configure the two DBMSes, and the results obtained by the test. In particular about the two best cases of each test, we have analyzed the operation system behavior to view the performance of the memory, disk, CPU and network components.

5.1 Result review

By analyzing the test results, we can highlight some very interesting results.

From the tests with MariaDB and the workload TPC-C, we can see in the table 5.1 that the best solutions are obtained with a standalone node. This result contradicts the initial hypothesis, namely that a balanced solution could gain a 80% throughput. Although it is slower than a solution in HA, the use of a balancer GLB has a gain of 9.04% over the synchronized solution, while the HAProxy balancer has an overhead of 4.86% over the same solution with the nodes synchronized.
If the goal of the customer was using two different hosts to improve the performance, the workload TPC-C shows that this solution does not satisfies this target. Although the goal is not met, you must consider the aspect of high reliability: to have an high-availability solution it is necessary to have at least two different nodes, and with the oldest solution the second node is always on the idles state and the fail-over time is more expensive then an load-balancing solution. Also with a traditionally active-passive solution, there are some problems with the machine maintenance, while with the balanced solution it is easyer because we can stop one node at time.
Then these tests show us that, with an overhead of 1.60%, there are two benefits: the possibility to maintenance the system without a disservice and real high-availability.

Table 5.1: MariaDB TPC-C result: gains and overhead for the best solution

Gain/overhead over base value						
Base Value	n1	n1+(n2)	n1+n2 glb	n1+(n2) glb	n1+n2 HAProxy	n1+(n2) HAProxy
n1	00.00%	-09.76%	-01.60%	-06.11%	-14.15%	-09.98%
n1 + (n2)	10.82%	00.00%	09.04%	04.04%	-04.86%	-00.25%
n1 + n2 + glb	01.63%	-08.29%	00.00%	-04.58%	-12.75%	-08.52%
n1 + (n2) + glb	06.51%	-03.89%	04.80%	00.00%	-08.56%	-04.12%
n1 + n2 + HAProxy	16.48%	05.11%	14.62%	09.36%	00.00%	04.85%
n1 + (n2) + HAProxy	11.09%	00.25%	09.31%	04.30%	-04.63%	00.00%

The tests with MariaDB and workload Epinions confirms some of the results obtained with the previous workload. The main one is that the solutions with the balancer HAProxy is slower than GLB. In fact there is an 6.93% of overhead between these solutions.
The second fact that can be seen on table 5.2, is that the two solutions that use a balancer are faster than the native solution active-active. The HAProxy solution has the 5.93% of gain over the standalone solution, while the GLB solution the 13.82% of gain. If the main goal of the customer is having an active-active configuration to achieve the objectives described above, the right solution indicated by these results is using a Galera load balancer.
This is the opposite result to those obtained with TPC-C workload, but you must consider that there is quite a difference between the two workload and the Epinions whom

goal is to simulate a real web portal.

Table 5.2: MariaDB Epinions result: gains and overhead for the best solution

Gain/overhead over base value						
Base Value	n1	n1+(n2)	n1+n2 glb	n1+(n2) glb	n1+n2 HAProxy	n1+(n2) HAProxy
n1	00.00%	-22.81%	13.82%	-05.55%	05.93%	-09.10%
n1 + (n2)	29.54%	00.00%	47.44%	22.35%	37.22%	17.76%
n1 + n2 + glb	-12.14%	-32.18%	00.00%	-17.01%	-06.93%	-20.13%
n1 + (n2) + glb	05.88%	-18.27%	20.50%	00.00%	12.15%	-03.76%
n1 + n2 + HAProxy	-05.60%	-27.13%	07.45%	-10.84%	00.00%	-14.19%
n1 + (n2) + HAProxy	10.01%	-15.08%	25.21%	03.90%	16.53%	00.00%

The PostgreSQL analysis is easyer than the MariaDB, because from the table 5.3 it is clear that the balancer *PgPool-II* does not meet the requirements of the customer. If we make a more detailed analysis, we can see that the gain of the solution where the balancer knows all hosts over the solution where the balancer knows a single host is about 0.70%. Clearly this result indicates that the balancer generates overhead. In fact the overhead from the solution without the balancer over the solution with the balancer is the 42.32%.
The unique positive aspect is that the overhead over the solution in cluster without the balancer is 7.62%, that is lesser than the overhead of MariaDB with the same workload. The two customer's goals are not completely satisfied with the PostgreSQL. The reason is simple: to have an high-availability solution, it is better that the nodes works together at the same time while with the PostgreSQL we are obliged to have one node active and one node passive. To manage the resources is necessary to adopt others software that can check the live state of the node and in case do a fail-over. In this case the Virtual IP address, that is local on the master machine, is moved to the node 2.
The second request of the customer is a solution that allows the maintenance of the machine without create a disservice and without hard procedures. To achieve this target PostgreSQL does not offers any solution, because when a fail-over happens on the replication mode, the secondary node becomes the first node, while the broken node leave the "cluster". The broken node can be re-joined to the cluster but the time necessary to do it may depending on the size of the database, and in any case is not automatically. This procedures involve risks and could become time consuming.

Table 5.3: PostgreSQL TPC-C result: gains and overhead for the best solution

Gain/overhead over base value				
Base Value	n1	n1+(n2)	n1+n2 PgPool-II	n1+(n2) PgPool-II
n1	00.00%	-07.62%	-46.72%	-47.09%
n1 + (n2)	08.25%	00.00%	-42.32%	-42.72%
n1 + n2 + PgPool-II	87.68%	73.37%	00.00%	-00.70%
n1 + (n2) + PgPool-II	89.00%	74.59%	00.70%	00.00%

The PostgreSQL with the workload Epinions, has the same results of the TPC-C workload. The overhead of the balancer solution is about the 70% and is greater than the overhead with the TPC-C workload. In this case the solution with two nodes is 4.83% worse than the single node, and this value is smaller than the previous workload. With this results it is clear that the balancer *PgPool-II* is not able to optimize the load, and the only effect that entails is the addition of a layer that increases the overhead.

Table 5.4: PostgreSQL Epinions result: gains and overhead for the best solution

Gain/overhead over base value				
Base Value	n1	n1+(n2)	n1+n2 PgPool-II	n1+(n2) PgPool-II
n1	00.00%	-04.38%	-70.93%	-70.48%
n1 + (n2)	04.58%	00.00%	-69.60%	-72.27%
n1 + n2 + PgPool-II	243.99%	228.93%	00.00%	-08.78%
n1 + (n2) + PgPool-II	277.10%	260.59%	09.63%	00.00%

From the tests with PostgreSQL we obtain an interesting results over the MariaDB. If we compare the throughput of the standalone solution with the Epinions workload, it turns out that PostgreSQL is 17% faster than MariaDB (9453.1874 VS 8033.9359). The outcome is the opposite with the TPC-C workload. In fact the PostgreSQL solution is worse than the 30% over MariaDB (760.3116 VS 1094.9105), and in accord of the analysis done over the Linux behaviour is clear that an intense activity of I/O degrades the performance of PostgreSQL.

5.1.1 Architecture and Tuning review

A background goal of this thesis is to understand how the I/O scheduler and the memory policy can change the performance of the DBMS. After that some differences of the results have been analyzed, we can state that:

- if the scheduler I/O is *CFQ* then is better avoid the memory swap of the system. The type of I/O load is indifferent because the swap use the disk layer and the *CFQ* policy has worse performance so it is recommended to avoid additional load of I/O with the use of the swap.

- if the scheduler I/O is *NOOP* then is the type of DBMS load that decides which is the best memory policy. The TPC-C workload has more disk I/O load of the Epinions, because the types of transaction are more complicated. Vice-versa, the transactions of Epinions are ”easyer” and do not require intermediate results to complete the task. So the memory policy that avoid the swap is better if the database is Web-Service oriented, while is better using the swap if there is an heavy-write load.

Everything considered, we can affirm that the best solution to satisfy the initial requirements is MariaDB in replication mode with the Galera Load Balancing. This is an high-availability solution that may improve the performance of the system.

PostgreSQL has better performance without the balancer, and when is used with the second node synchronized is a valid alternative to MariaDB, because offers a zero data lost solution with a minimal time of disservices when the fail-over of the node 1 happens.
If the DBMS is used by the web-applications and the performance is the first aim, then the DBMS that must be chosen is PostgreSQL.

5.2 Experience gained

At the end of the work, there are some new notions that I have learned besides the results of this essay. My database's background has increased, because I have deepened how the PostgreSQL and MariaDB work, and generally speaking the theory of the DBMS. I studied the benchmark of databases, discovering a great tool, *oltpbenchmark*, and the different types of workload that the literature offers. What surprised me was the gain we have obtained with the system optimization, an effort the we have to make even nowadays: obviously it is always highly recommended to invest a bit of time to systems tuning. In particular, the tuning of Linux systems is an activity currently entrusted but this thesis emphasize that we must not underestimate it or take it for granted.
Finally, I have learned which open source balancers exist, and which features they have. I also confirmed my personal idea that most of the time they are just a bottleneck, and if there are hardware balancer there is a good reason for that.

5.3 Bugs opened

Working on this thesis, I have utilized many software for the first time. It is my habit to try to deepen and thoroughly understand the software that I use, and thanks to this principle I could make a small contribution to the open source community by reporting and fixing some minor bugs. These are:

- GLB: the installation of GLB includes a script to manage the balancer as a daemon. This script had many logical errors with many variables reversed. In addition to these minor problems, the most serious error was how the balancer was invoked. I Have created a patch that has been deployed [glb].
- PgPool-II: the script to do the setup, checked if the prerequisite software was installed on the system. This control was commented on the stable version and the BUG [pgpa] has be signaled.
- oltpbenchmark: The most serious bug i have fixed on this thesis regards the benchmark used. This bug has been discovered because when i started the test with the Epinions on PostgreSQL, the throughput was about the 22 req/sec versus the 8000 of MariaDB. This result was very odd and once analyzed I found that the definition of the two databases was different. In fact on the PostgreSQL the definition was missing the declaration of three index that MariaDB had. This bug has been signaled and fixed [olta].

5.4 Future works

This work started out with this goal: to find out which high-availability solution exists over the open-source database management system and understand which is the best solution in terms of performance.

Below follows a list of topics that should be of interest for further investigations:

- Once verified that the solution with more nodes has worse performance over the standalone solution, Is it possible to reduce the latency over the nodes to increase throughput? Might be interesting to look for new solutions to increase the efficiency of communication between nodes, perhaps through a quorum disk or through the fibre channel communication.
- I have tested the two major solution open-source, but there are other interesting open-source DBMS such *SQLite*, *Firebird*, *Ingres*. Are these DBMS faster that the two major?

- could be interesting to test the Linux's tuning behave over the closed-source DBMS, and compare the performance of the two types.
- The classic high-availability solution is implemented with two nodes but how the system behaves if others nodes are added? The performances improve or decrease?
- I have executed the Linux's tuning of two parameters, but there are some others optimization that can improve the performance. Another further work could be to investigate on the possible optimization of the operating system.
- Today, there are some company that need to have a database replicated on multi site connected with a fast connection. Is the latency of this link sufficient to implement this type of solution or would be better to have an asynchronous replication mode?

A

Some technicalities

Listing A.1: Haproxy configuration's file

```
global
    log         127.0.0.1 local2
    chroot      /var/lib/haproxy
    pidfile     /var/run/haproxy.pid
    maxconn     200
    user        haproxy
    group       haproxy
    daemon
    stats socket /var/lib/haproxy/stats mode 0600 level admin

defaults
        log     global
        mode    tcp
        option  tcplog
        option  dontlognull
        retries 3
        option redispatch
        maxconn 200
        timeout connect 5000ms
        timeout client 50000ms
        timeout server 50000ms

frontend db_write
        bind 192.168.2.105:3306
        default_backend cluster_mariadb

backend cluster_mariadb
        mode tcp
        option tcpka
        balance roundrobin
        option mysql-check user haproxy
        server dbo01 192.168.2.111:3306 check weight 1
        server dbo02 192.168.2.112:3306 check weight 1
```

Listing A.2: GLB configuration's file

```
LISTEN_ADDR="192.168.2.105:3306"
CONTROL_ADDR="127.0.0.1:8011"
THREADS="8"
MAX_CONN="200"
DEFAULT_TARGETS="192.168.2.111:3306␣192.168.2.112:3306"
OTHER_OPTIONS="-w␣exec:'/opt/gian/glb_mysql.sh␣-d␣2␣-uroot␣-
   pthepassword'"
```

Listing A.3: GLB script to check the liveness of node

```
#!/bin/sh -u
#
# Copyright (C) 2012 Codership Oy <info@codership.com>
# This is a script for glbd exec watchdog backend to monitor the
    availability
# of MySQL server.

ADDR=$1; shift

DONOR_STATE=2
if [ "$1" = "-d" ]; then
    shift; DONOR_STATE=$1; shift
fi

HOST=$(echo $ADDR | cut -d ':' -f 1); HOST=${HOST:-"127.0.0.1"}
PORT=$(echo $ADDR | cut -s -d ':' -f 2); PORT=${PORT:-"3306"}

QUERY="SHOW␣STATUS␣LIKE␣'wsrep_local_state';␣SHOW␣STATUS␣LIKE␣'
   wsrep_incoming_addresses'"

while read CMD; do
    [ "$CMD" != "poll" ] && break;
    RES=$(mysql -B --disable-column-names -h$HOST -P$PORT $* -e "
       $QUERY")
    if [ $? -eq 0 ]; then
        STATE=$(echo $RES | cut -d ' ' -f 2)
        OTHERS=$(echo $RES | cut -d ' ' -f 4)
        STATE=${STATE:-"4"}
    else
        STATE=
        OTHERS=
    fi

    # convert wsrep state to glbd code
    case $STATE in
        4) STATE="3"
        ;;
        3) STATE="2"
```

```
        ;;
        2) STATE="$DONOR_STATE"
        ;;
        1|5) STATE="1"
        ;;
        0|*) STATE="0"
    esac
    echo "$STATE␣$OTHERS"
done
```

Listing A.4: PgPool-II configuration's file

```
listen_addresses = '*'
port = 9999
socket_dir = '/tmp'
pcp_port = 9898
pcp_socket_dir = '/tmp'
enable_pool_hba = off
pool_passwd = 'pool_passwd'
authentication_timeout = 60
ssl = off
num_init_children = 110
max_pool = 10
child_life_time = 30
child_max_connections = 0
connection_life_time = 30
client_idle_limit = 0
log_destination = 'stderr'
print_timestamp = on
log_connections = off
log_hostname = off
log_statement = off
log_per_node_statement = off
log_standby_delay = 'none'
syslog_facility = 'LOCAL0'
syslog_ident = 'pgpool'
debug_level = 0
pid_file_name = '/var/run/pgpool/pgpool.pid'
logdir = '/var/log/pgpool'
connection_cache = on
reset_query_list = 'ABORT; DISCARD ALL'
replication_mode = off
replicate_select = off
insert_lock = on
lobj_lock_table = ''
replication_stop_on_mismatch = off
failover_if_affected_tuples_mismatch = off
load_balance_mode = on
ignore_leading_white_space = on
white_function_list = ''
```

```
black_function_list = 'nextval,setval'
master_slave_mode = on
master_slave_sub_mode = 'stream'
sr_check_period = 0
sr_check_user = 'postgres'
sr_check_password = 'thepassword'
delay_threshold = 0
follow_master_command = 'pg_ctl promote'
parallel_mode = off
pgpool2_hostname = ''
system_db_hostname = 'localhost'
system_db_port = 5432
system_db_dbname = 'pgpool'
system_db_schema = 'pgpool_catalog'
system_db_user = 'pgpool'
system_db_password = ''
health_check_period = 0
health_check_timeout = 20
health_check_user = 'postgres'
health_check_password = 'thepassword'
health_check_max_retries = 0
health_check_retry_delay = 1
failover_command = ''
failback_command = ''
fail_over_on_backend_error = on
search_primary_node_timeout = 10
recovery_user = 'nobody'
recovery_password = ''
recovery_1st_stage_command = ''
recovery_2nd_stage_command = ''
recovery_timeout = 90
client_idle_limit_in_recovery = 0
use_watchdog = off
trusted_servers = ''
ping_path = '/bin'
wd_hostname = ''
wd_port = 9000
wd_authkey = ''
delegate_IP = ''
ifconfig_path = '/sbin'
if_up_cmd = 'ifconfig eth0:0 inet $_IP_$ netmask 255.255.255.0'
if_down_cmd = 'ifconfig eth0:0 down'
arping_path = '/usr/sbin            # arping command path'
arping_cmd = 'arping -U $_IP_$ -w 1'
clear_memqcache_on_escalation = on
wd_escalation_command = ''
wd_lifecheck_method = 'heartbeat'
wd_interval = 10
wd_heartbeat_port = 9694
```

```
wd_heartbeat_keepalive = 2
wd_heartbeat_deadtime = 30
wd_life_point = 3
wd_lifecheck_query = 'SELECT 1'
wd_lifecheck_dbname = 'postgres'
wd_lifecheck_user = 'postgres'
wd_lifecheck_password = 'thepassword'
relcache_expire = 0
relcache_size = 256
check_temp_table = on
memory_cache_enabled = off
memqcache_method = 'shmem'
memqcache_memcached_host = 'localhost'
memqcache_memcached_port = 11211
memqcache_total_size = 67108864
memqcache_max_num_cache = 1000000
memqcache_expire = 0
memqcache_auto_cache_invalidation = on
memqcache_maxcache = 409600
memqcache_cache_block_size = 1048576
memqcache_oiddir = '/var/log/pgpool/oiddir'
white_memqcache_table_list = ''
black_memqcache_table_list = ''
ssl_key = ''
ssl_cert = ''
ssl_ca_cert = ''
ssl_ca_cert_dir = ''
backend_hostname0 = '192.168.2.111'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/opt/postgres/data'
backend_flag0= 'DISALLOW_TO_FAILOVER'
other_pgpool_hostname0 = ''
other_pgpool_port0 =
other_wd_port0 =
heartbeat_destination0 = 'host0_ip1'
heartbeat_destination_port0 = 9694
heartbeat_device0 = ''
backend_hostname1 = '192.168.2.112'
backend_port1 = 5432
backend_weight1 = 1
backend_data_directory1 = '/opt/postgres/data'
backend_flag1= 'DISALLOW_TO_FAILOVER'
```

Bibliography

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[bsd] Bsd license. `http://en.wikipedia.org/wiki/BSD_licenses`.

[CDKB11] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Pearson Education, 2011.

[CK88] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. 1988.

[Cod] Codership. Galera cluster for mysql. `http://codership.com/content/using-galera-cluster`.

[Com98] International Electrotechnical Commission. *Functional Safety of Electrical/electronic/programmable Electronic Safety-related Systems: Sécurité Fonctionnelle Des Systèmes Électriques/électroniques/électroniques Programmables Relatifs À la Sécurité. Partie 2, Prescription Pour Les Systèmes Électriques/électroniques/électroniques Programmables Relatifs À la Sécurité. Requirements for electrical/electronic/programmable electronic safety-related systems*. Number pt. 2. International Electrotechnical Commission, 1998.

[Cor] Microsoft Corporation. Microsoft sql server 2012. `http://www.microsoft.com/sqlserver/2012/`.

[Cri91] Flaviu Cristian. *Understanding fault-tolerant distributed systems*. Communications of the ACM, 1991.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[GHOS96] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. 1996.

[glb] Bug of glb. `https://groups.google.com/d/msg/codership-team/mcZfawBCXI0/_7oyU2ILLXEJ`.

[gpl] Gnu general public license. `http://www.gnu.org/licenses/gpl.html`.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations, 1981.

[Gra85] Jim Gray. Why do computers stop and what can be done about it?, 1985.

[Gra90] Jim Gray. A census of tandem system availability between 1985 and 1990. IEEE, 1990.

[Haa01] Peter Haase. Heterogeneous data replication. 2001.

[hap] Haproxy website. `http://haproxy.1wt.eu/#desi`.

[Hat] Red Hat. Red hat enterpise linux. `http://www.redhat.com/products/enterprise-linux/`.

[HT93] Vassos Hadzilacos and Sam Toueg. Distributed systems (2nd ed.). chapter Fault-tolerant Broadcasts and Related Problems. 1993.

[IBMa] IBM. Ibm db2 for z/os. `http://www-01.ibm.com/software/data/db2/zos/db2-10-zos/index.html`.

[IBMb] IBM. Ibm system storage ds8000. `http://www-03.ibm.com/systems/storage/disk/ds8000/index.html`.

[IBMc] IBM. Ibm system storage san volume controller. `www.ibm.com/systems/storage/software/virtualization/svc`.

[IEEa] IEEE. Ieee p802.3ad link aggregation task force. `http://grouper.ieee.org/groups/802/3/ad/index.html`.

[IEEb] IEEE. Ieee std. 802.1q-2011. `http://standards.ieee.org/getieee802/download/802.1Q-2011.pdf`.

[Kea09] B. Challenges Keating. Involved in multimaster replication. `http://www.dbspecialists.com/files/presentations/mm_replication.html`, 2009.

[Lap92] J.C. Laprie. *Dependability: basic concepts and terminology*. Dependable computing and fault-tolerant systems. Springer-Verlag, 1992.

[MA05] Paolo Massa and Paolo Avesani. Controversial users demand local trust metrics: An experimental study on epinions.com community. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, AAAI'05, pages 121–126. AAAI Press, 2005.

[mara] Mariadb foundation. `http://en.wikipedia.org/wiki/Mariadb#MariaDB_Foundation`.

[marb] Mariadb galera cluster 5.5.29 - release note. `https://blog.mariadb.org/mariadb-galera-cluster-5-5-29-stable-ga-released`.

[mys] Mysql 5.1 - release note. `http://dev.mysql.com/doc/relnotes/mysql/5.1/en/index.html`.

[olta] Bug of oltpbenchmark. `http://www.pgpool.net/mantisbt/view.php?id=78`.

[oltb] Oltpbenchmark website. `http://oltpbenchmark.com`.

[oraa] Oralce comporation buy sun microsystems. `http://en.wikipedia.org/wiki/Sun_acquisition_by_Oracle`.

[Orab] Oracle. Mysql cluster. `http://www.mysql.com/products/database/cluster`.

[Orac] Oracle. Oracle real application cluster. `http://www.oracle.com/us/products/database/options/real-application-clusters/overview/index.html`.

[ÖV11] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Computer science. Springer, 2011.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Inc., New York, NY, USA, 1986.

[PCZ02] R.B.M. Paul C. Zikopoulos. *Db2: The Complete Reference*. McGraw-Hill Education (India) Pvt Limited, 2002.

[Per] Percona. Percona xtradb cluster. =http://www.percona.com/software/percona-xtradb-cluster.

[PG] Fernando Pedone and Rachid Guerraoui. On transaction liveness in replicated databases.

[pgpa] Bug of pgpool. `https://github.com/oltpbenchmark/oltpbench/issues/72`.

[pgpb] Pgpool-ii. http://www.pgpool.net/mediawiki/index.php/Main_Page.

[posa] Postgres 8.4 - release note. http://www.postgresql.org/docs/8.4/static/release-8-4-19.html.

[posb] Postgres-xc. http://sourceforge.net/apps/mediawiki/postgres-xc/.

[posc] Postgresql 9.1 - release note. http://www.postgresql.org/docs/current/static/release-9-1.html.

[RR96] R. Riedl and L. Richter. Classification of load distribution algorithms. In *Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96)*, Washington, DC, USA, 1996. IEEE Computer Society.

[Sto86] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[VMW] VMWare. Vmware vshere, ha cluster. http://www.vmware.com/products/vsphere/features-high-availability.