Performance Evaluation of ZooKeeper Atomic Broadcast Protocol

Said Naser Said Kamil[1(✉)], Nigel Thomas[2], and Ibrahim Elsanosi[3]

[1] Faculty of Science-Khoms, Elmergib University, Khoms, Libya
said.kamil@elmergib.edu.ly
[2] School of Computing, Newcastle University, Newcastle upon Tyne, UK
nigel.thomas@ncl.ac.uk
[3] Faculty of Science, Sebha University, Sebha, Libya
i.elsanosi@sebhau.edu.ly

Abstract. In this paper, we present a performance model of the Zab protocol formally specified using the Markovian process algebra PEPA. The model is parameterised from measurements taken from a real deployment of Zookeeper and is evaluated to derive estimates for average latency and throughput at various loads. These estimates are then compared against further measurements from the real system. Although the model is highly abstract and ignores much implementation detail, it is shown to give qualitative predictions for system behaviour, most notably for estimating the saturation point.

Keywords: Cloud computing · Performance evaluation · ZooKeeper · Zab protocol · Scalability · PEPA

1 Introduction

Large scale systems have witnessed dramatic increases for high performance, availability and coordinations. More recently, ZooKeeper has proposed a system for high performance and availability coordination service. ZooKeeper is a distributed coordination service for cloud computing applications and it is providing fundamental coordination primitives, for instance, synchronisation and group services for distributed applications [1,15,16]. Additionally, ZooKeeper is an open source Apache project providing a fault-tolerant coordination service and it is maintained by Apache Software Foundation and Yahoo. A key feature of ZooKeeper is offering high-performance services and coordination of the large scale systems. Cloud computing applications also, can leverage fundamental services that provided by ZooKeeper through the encapsulation of the distributed coordination algorithms and preserving a simple database [20,26]. According to Hunt et al. [15], "ZooKeeper can handle tens to hundreds of thousands of transactions per second". Indeed, ZooKeeper is used as a standard coordination kernel for several distributed systems, for instance, Facebook, Yahoo, Netflix

Q. Zhao and L. Xia (Eds.): VALUETOOLS 2021, LNICST 404, pp. 56–71, 2021.
https://doi.org/10.1007/978-3-030-92511-6_4

and Twitter [4,9,25]. Furthermore, a high availability is provided by ZooKeeper through the replication of data on each server in an ensemble of servers.

A collection of services is provided by the ZooKeeper; group services, configuration and synchronisation are provided in a form of centralised coordination service [1]. An additional characteristic offered by ZooKeeper interface is wait-free data objects [15] that provide a simple and powerful coordination service. ZooKeeper not only guarantees the ordering FIFO for the execution of clients' requests, but also guarantees linearizability for all write requests that update the state of the ZooKeeper. ZooKeeper guarantees liveness and durability of services. The design of ZooKeeper uses some ideas from previous coordination services, for example, Chubby [5] and Boxwood [19]. ZooKeeper does not use locks in its API, but allows a client to use them. While Chubby only allowed the client to connect to the leader, ZooKeeper allows a client to connect to any server in an ensemble (leader and/or followers). Therefore, ZooKeeper provides more flexibility and higher performance requirements.

The performance of ZooKeeper atomic broadcast protocol Zab will be examined using the PEPA Eclipse Plug-in tool [24]. Specifically, the broadcast phase will be modelled and an approximation of its behaviour will be shown taking into account several performance metrics (Latency and Throughput). Furthermore, the model will be presented considering the cyclic behaviour of the system, by means of the scalable analysis provided by the Eclipse Plug-in tool using Ordinary Differential Equations (ODEs) to derive measures of throughput and latency.

The outline of this paper is as follows. An overview for the ZooKeeper is given in the next section. Then the atomic broadcast protocol (Zab) is presented. This is followed by an illustration of the Zab PEPA model. Next, the experiments and results are shown and discussed. Finally, the paper ends with some conclusions and future work.

2 ZooKeeper

ZooKeeper service consists of an ensemble of servers, which use replication in order to attain high performance and availability. Several functions are provided through the ZooKeeper API, (create, delete, exists, getData, setData, getChildren and sync); these methods have both synchronous (an application use it in the case of executing a single operation) and asynchronous (can be used for both executing tasks and for multiple outstanding operations). That is to say, using an asynchronous operation technique allows the client to have multiple outstanding operations at one time. ZooKeeper uses the atomic broadcast protocol (Zab) to guarantee linearizability and the watches mechanism to allow clients to watch any updates and receive notification about their data object without requiring polling. Furthermore, ZooKeeper provides data nodes (znodes) in a hierarchical namespace that allow clients to manipulate these data objectives while taking into account a set of primitives, for instance, group membership and configuration management (for details, see [15]). A new approach in the coordination is

presented through implementing the powerful primitives at the client side using the ZooKeeper API, which will provide a high performance. Figure 1 displays the high level components of the ZooKeeper service.

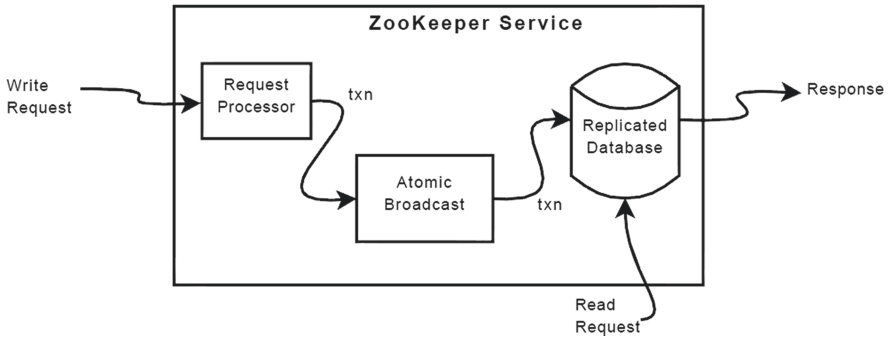

Fig. 1. Components of the ZooKeeper service [15]

As ZooKeeper is comprised of an ensemble of servers, it uses majority quorums, which means that it will proceed only when the majority of servers work and are consistent. Servers can crash and recover many times without affecting the service as long as Zab has a majority. In the case of leader failure, Zab controls the election of a new leader and guarantees the continuity of service. The leader replicates requests consistently to followers, thus eventually they finish up with the same state. Additionally, timeouts have been used in order to detect failures. For recovering from faults, such as, power outages, it is essential to record messages on the disk media of a quorum of followers before the delivery process takes place [21]. Moreover, ZooKeeper uses a database in-memory and stores periodic snapshots and transactions log on disk, as writing every single transaction on the disk would result in the disk I/O bottleneck [21].

Clients can connect to one ZooKeeper server to submit its requests. Read requests are serviced from a local replica of the database of the contacted server, this provides high read throughput. However, write requests that change the state of the service, are transformed to idempotent transactions and forwarded to the leader using Zab protocol, thus write throughput is limited by the throughput of Zab protocol [21]. Furthermore, the server that received the client request, will respond to that client with the corresponding state update.

3 Atomic Broadcast Protocol Zab

ZooKeeper uses the atomic broadcast protocol (Zab) for the coordination of write requests that update the state of ZooKeeper service. In short, Zab is a high-performance protocol for primary-backup systems [16]. Zab provides efficient mechanisms for process recovery, where it enables multiple outstanding state

changes [16]. Additionally, Zab keeps the replicas of the ZooKeeper state at each server consistent [21]. Furthermore, Zab protocol has two modes: broadcast and recovery [21], where it transitions to a recovery mode once the service starts or after a failure of a leader. There are two requirements to end the recovery mode: a leader emerges and then synchronising the state of a quorum of servers with the leader. The leader can start broadcasting once the synchronisation phase successfully finished, the broadcast mode ends if the leader fails or it lost the support of a quorum of followers.

The Zab differs from the other atomic broadcast protocols, such as Paxos [18], whereas Zab enables multiple outstanding operations and guarantees the execution order of the clients' requests. ZooKeeper maintaining ordered broadcast FIFO using Zab, where it guarantees to deliver the transactions in the same order to all servers, and guarantees that transactions cannot be skipped by servers [17]. ZooKeeper developers have proposed a Primary Order property that maintains the right order of the state change over time, which is extremely important for the primary-backup systems. As stated by Hunt *et al.* [15], the implementation of Zab showing that it can meet the high performance and low latency demands of broadcasts. Additionally, the TCP channel connections are used for the message exchange between servers (Zab processes), and it has been relied on the nature of the TCP to maintain the right order.

According to Junqueira *et al.* [16], Zab consists of three phases: discovery, synchronisation, and broadcast. There are two types of servers (Leader and Follower), which can be performed by Zab process. Phase 1, is used to determine which server is the leader and phase 2 to establish the leadership and phase 3 is the broadcast phase. In this work, we will only investigate the Zab broadcast phase, however, for more details about the phases 1 and 2 we refer the interested reader to [16].

As stated by Junqueira and Reed [17], there are additional checks that need to be performed by the follower before the acknowledgement of a proposal take place. Firstly, checking that the proposal is from the current leader that the follower f is following. Secondly, checking the order of proposals and transactions to be in the same order, as the leader broadcasts them.

4 Zab PEPA Model

A PEPA model has been created for the Zab protocol, specifically, the broadcast phase using the Performance Evaluation Process Algebra (PEPA) modelling paradigm. The primary objective is to inspect and evaluate the behaviour of the Zab protocol (broadcast phase). Validating the outcomes of the model against a real implementation of Zab is another objective. PEPA is a high-level quantitative modelling language developed by Hillston [13], and one of its essential usage is for modelling the distributed systems. Models built on a high level of abstraction using stochastic process algebra (SPA), for instance (PEPA [13], EMPA [3] and SPADES [10]), and stochastic Petri net (SPN) [7]. PEPA offers several significant features in performance modelling, such as compositionality, formality

and abstraction [14]. The PEPA Eclipse Plug-in tool [24] is a supporting tool has been used for developing and analysing the performance of systems, offering a variety of analysis techniques, for example, continuous time Markov chain (CTMC), Stochastic Simulation and Ordinary Differential Equations (ODEs). Such approaches allowing the observation of a system as it evolves from an initial state over a period of time [24]. Furthermore, each action within the PEPA model has a rate which is the reciprocal of the average duration, or delay, to undertake by the action. Each of the model components has been modelled as a set of sequential actions to simulate the protocol behaviour. Furthermore, several performance metrics, for instance, throughput and latency have been used to evaluate the performance of the Zab PEPA model.

The model is analysed using Steady State Analysis CTMC, and also, its behaviour is approximated using the Ordinary Differential Equations (ODEs) analysis for both throughput and population, which supports the numerical calculation of a large scale model with a large number of *Client* and *Request* components. ODE solvers are continuous and deterministic [24] and have been used by [12] and [11] as a solution to the CTMC state space explosion problem. The number of clients and requests has been varied (from 1 to 10), every single client generates (20) threads and sends requests to the *Request* component. Accordingly, the number of requests that sent to a server equals to the number of clients times (20) threads (e.g. 10 clients * 20 threads = 200 requests sent in parallel). In order to model the Zab protocol in PEPA, several simplifications have been applied with the objective to get a scalable model, that can process a large number of requests, and also, to allow to explore further implications of some actions (e.g. cpu action). The Zab broadcast PEPA model is represented in an abstraction level with the following modifications: Firstly, it has been created *Request* component which has a sequence of all actions that specified by the Zab protocol. So, the *Request* component has been used in the model to simulate the overall performance, i.e. allowing to evaluate the overload of the *Leader* in another way. Additionally, the other components have been used in the model are (*Client* and *Leader*). Whereas, the actions of these components are manipulated in parallel. In other words, these components are simultaneously cooperating their actions with the primary component (*Request*). As well, the *Client* component is used to represent a client who sends a write request or update existing data.

Furthermore, the sequence of the actions is preserved through the *Request* component. These changes not only simplify the model, but also, allow the model to scale up without any restrictions. Hence, the overall performance will not be affected, i.e. quantitative analysis. Moreover, the model follows a regularly repeated sequence of events. For example, a client generates a request, wait for processing this request and once the processed request is received (i.e. *getRequest* action) by the client then the model returns to the initial state (i.e. generate request) to send a new request and so on, i.e. which is a cyclic model.

Some local actions of the *Leader*, such as, (*excuteRequest, processingCommit* and *leaderDeliver*) have been renamed to *cpu* action in the model and the rate

of the *cpu* action equals to the average duration rate of these three actions. Whereas, we consider two different options for the interaction between the leader and the clients in ZooKeeper. Namely, we have made an assumption that the model will be examined into two different versions, where the action *leaderDeliver* will be used as an independent action, and then it will be used as a centralised action. Firstly, we have used the model with (2 cpu) actions (*excuteRequest and processingCommit*), and will be referred to as (*Assumption 1*). Secondly, it has been analysed by using (3 cpu) actions, where added the action (*leaderDeliver*), which is the action of the component $Request_{10}$. The main reason for using a different number of *cpu* actions is to examine the model with various performance scenarios and to explore which of these actions will give rise to overload the leader. Additionally, the *cpu* action is used because PEPA does not allow to directly limit the action across multiple actions, i.e. limits the rate across multiple actions. That is to say, it will be a single action (*cpu*) but we are limiting the total rate of this action, it is a combination of all those actions. So, the *cpu* cannot run faster than those all actions (bounded capacity).

The following is the Zab PEPA model:

$$
\begin{aligned}
Client &\stackrel{def}{=} (generateRequest, r_1).(getRequest, r_1).Client \\
Request &\stackrel{def}{=} (generateRequest, r_1).Request_1 \\
Request_1 &\stackrel{def}{=} (requestToL, r_1).Request_2 \\
Request_2 &\stackrel{def}{=} (cpu, c).Request_3 \\
Request_3 &\stackrel{def}{=} (sndProposalA, r_4).Request_4 \\
Request_4 &\stackrel{def}{=} (sndProposalB, r_4).Request_5 \\
Request_5 &\stackrel{def}{=} (processingAcknowledgeA, r_9).Request_6 \\
&\quad + (processingAcknowledgeB, r_9).Request_6 \\
Request_6 &\stackrel{def}{=} (acknowledgeA, r_5).Request_7 \\
&\quad + (acknowledgeB, r_5).Request_7 \\
Request_7 &\stackrel{def}{=} (cpu, c).Request_8 \\
Request_8 &\stackrel{def}{=} (commitA, r_7).Request_9 \\
Request_9 &\stackrel{def}{=} (commitB, r_7).Request_{10} \\
Request_{10} &\stackrel{def}{=} (leaderDeliver, r_8).Request_{11} \\
Request_{11} &\stackrel{def}{=} (getRequest, r_1).Request \\
Leader &\stackrel{def}{=} (cpu, c).Leader \\
System &\stackrel{def}{=} Client[N] \underset{L_1}{\bowtie} Request[N] \underset{L_2}{\bowtie} Leader
\end{aligned}
$$

The components of the model are cooperated concurrently as exhibited in the system equation, where *Client* cooperate with *Request* over the set L_1. Indeed, the *getRequest* action has been used to restrict a client who sent a request from sending a new request until get a response, consequently, a client will be able to send a new request. Also, the *Request* and *Leader* cooperate over the set L_2. Whereas, N has been varied from 20 to 200, and the cooperation set $L_1 = \{generateRequest, getRequest\}$ and the set $L_2 = \{cpu\}$.

Table 1. The Rates of Zab PEPA model

Rate	Value	Rate	Value
r_1	7.42	r_3	61.96657616
r_4	7.42	r_5	7.42
r_6	437.1975906	r_7	7.42
r_8	26.33486286	r_9	1330.250132
c	3/(1/r3+1/r6+1/r8)		

It is worth noting that, the model parameters representing a real-time implementation, which is in milliseconds. However, the rates of *r1, r4, r5 and r7* are not part of the real measurement of the system. That is because of the well-known problem in the real-time distributed systems (clock synchronisation). In fact, there are many solutions for the clock synchronisation problem, but they have not been augmented in the system that we have taken the measurements from. Therefore, we have calculated those rates as follows:

$$rate_i = \frac{L + \sum r_i}{N} \tag{1}$$

Where, L is the average latency of the real measurements, r_i representing the real time in ms for the actions associated with these rates (r_3, r_6, r_8 and r_9). Also, N is representing the repetition of times that the rates (r_1, r_4, r_5 and r_7) have been used in the model. The rate of each action equals $(1/Time(ms))$ as shown in the Table 1.

5 Performance Metrics

Two performance benchmarks will be considered (latency and throughput) with the intention of inspecting the model behaviour and to compare the PEPA model with the real implementation of Zab protocol.

Latency is a performance benchmark, whereby it means the total time for a data transmitted to a destination and then returned back to its source (measuring the round trip). In other terms, it used to refer to a delay. Based on the derived ODEs analysis results the latency has been calculated as follows:

– **Assumption 1:**

$$(1 + Pop(Req2 + Req7))\frac{2}{c} + \sum_{\forall a \in \sigma} \frac{1}{rate_a} \tag{2}$$

– **Assumption 2:**

$$(1 + Pop(Req2 + Req7 + Req10))\frac{3}{c} + \sum_{\forall a \in \sigma} \frac{1}{rate_a} \tag{3}$$

Where $\sigma = \{generateRequest,\ requestToL,\ sndProposalA,\ sndProposalB,\ processing\ AcknowledgeA,\ processingAcknowledgeB,\ acknowledgeA,\ acknowledgeB,\ commitA,\ commitB\ and\ getRequest\}$. Also, a is the type action for all actions in the set σ. It is clear that the model here is a closed queuing network. The reason of calculating the latency for the non *cpu* actions $1/rate$ because they are not subject to queuing, just the time it takes to do that action. But for the *cpu* actions, they have to queue (competitive actions) and they take serving time and the average waiting time. Hence, the population average queue length equals 1 plus the queue length for 1 less entity in it, so the population is for the system with $N - 1$ requests.

Throughput is the second performance metric used. It has been calculated using the PEPA Eclipse Plug-in tool scalable analysis (Throughput), which gives how many times a specific action happen in a millisecond.

6 Experiments and Results

In this section, a number of assumptions have been made to analyse the protocol, for instance, all experiments are processed in the absence of failure and the latency is only measured from the server side. The model which is illustrated in the Sect. 4 has been analysed and evaluated using PEPA Eclipse Plug-in tool, i.e. steady state analysis (CTMC) and the fluid flow analysis (ODEs). Basically, the time which is used in the model and illustrated in the Table 1, is taken from a real implementation of the Zab protocol. Specifically, it is representing the average of three implementations of Zab for each number of clients. However, in the experiments which follow the rates are calculated by taking the average of those times, which means that the rates that have been used in the Zab PEPA model are representing the average of (30) experiments for the number of clients (from 1 to 10). As each client generates (20) threads, in our model number of clients have been varied (from 20 to 200) in order to simulate the number of threads for each number of clients.

In the first set of experiments, we have analysed the model by the means of CTMC (i.e. compare *Assumption 1* and *Assumption 2*), and then the Zab performance (*Assumption 1* and *Assumption 2*) have been evaluated using the ODEs. Also, we have investigated the performance issue that arises in the *Assumption 2*. Finally, it has been compared the latency of Zab *Assumption 1* against the real implementation of Zab.

6.1 CTMC Analysis

The results of the CTMC analysis will be displayed in this section. The main reason for using the CTMC is to figure out the scale of the model, specifically, the number of clients and requests that can be analysed with this type of analyses. The following figures illustrate the outcomes of the model *Assumption 1* and *Assumption 2*, in terms of performance metrics (throughput and the populations), where the maximum number of clients can be derived using the CTMC

is (4). As seen in Fig. 2, the throughput of *cpu* action of the *Assumption 2* is higher than the *Assumption 1* that is because we have used 3 *cpu* action in the *Assumption 2* and the number of clients and requests is small. So, the leader has enough resources to manipulate this loads. However, the throughput of the *getRequests* action of both *Assumption 1* and *Assumption 2* are identical, due to the use of small load (i.e. limited by the CTMC state space problem). In Fig. 3, the population of *Assumption 1* and *Assumption 2* are showing the same behaviour, and also, there is a very small waiting time. It is clearly that the utilisation is very low, as the CTMC only allowed us to analyse the model using (4 clients) before the state space problem arises.

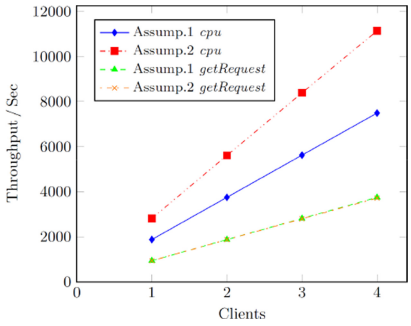

Fig. 2. Throughput of Zab (Assump.1 & Assump.2) using CTMC (*cpu* and *getRequest* actions).

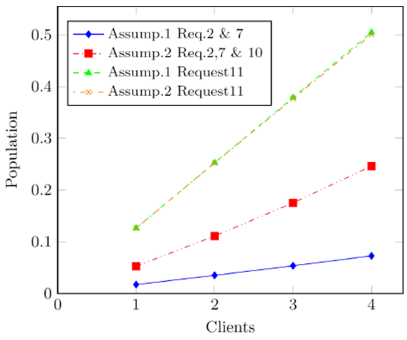

Fig. 3. Population of Zab (Assump.1 and Assump.2) using CTMC (Request2,7,10 and Request11).

6.2 Zab PEPA Model ODEs Analysis

The latency and the throughput of the Zab protocol PEPA models *Assumption 1* and *Assumption 2* are shown respectively in Fig. 4 and Fig. 5. Obviously, *Assumption 1* of Zab performs much better than the Zab *Assumption 2* in both latency and throughput. In Fig. 4 the latency of the Zab *Assumption 1* is displayed as a flat line at the beginning up to the point where $N = 60$. Then it has increased gradually to reach its maximum latency of 6.2538 ms, when the number of clients equals 200. This means that the Leader has the capacity to handle the coming requests up to $N = 60$, then the queuing time will start to increase with the increased load on the system. On the other hand, the latency of Zab *Assumption 2* is linearly increased from the very beginning and it is continued up to the end with the increase of the load, where the latency is extremely high.

It is obvious that introducing the extra *cpu* action indeed affects the scalability; overloading the Leader and saturation is reached with a very low number of clients. With 2 *cpu* actions that saturation does not occur until 60 clients,

and shows a more scalable approach. These because the load on the Leader is not so great the latency does not rise as significantly.

The throughput shown in Fig. 5 illustrates that the performance of Zab *Assumption 2* is poor and it is just shown as a flat line. However, Zab *Assumption 1* presents a higher throughput, where it reaches its peak when the number of clients is 60 (i.e. 54273 requests per second). The poor performance exemplified by the Zab *Assumption 2* (Fig. 5) is because the system is saturated where the use of 3 *cpu* results in more contention on the resources. Hence, the rate which is used for the *cpu* action (c) is much lower than the same rate that has been used for the *Assumption 1* for the same action. The average duration rate (c) for the *Assumption 1* is calculated as:

$$c_1 = \frac{2}{\frac{1}{r_3} + \frac{1}{r_6}} \tag{4}$$

For the *Assumption 2*, it is calculated as:

$$c_2 = \frac{3}{\frac{1}{r_3} + \frac{1}{r_6} + \frac{1}{r_8}} \tag{5}$$

As $1/r_3 + 1/r_6 < 2/r_8$ using the parameters in Table 1, the value of c used in *Assumption 2* is less than in *Assumption 1*. In fact c is less than half as fast in *Assumption 2* than *Assumption 1*.

Thus, the performance of the model is significantly affected. Therefore, in the next section we will investigate further the behaviour of the Zab *Assumption 2* in order to tackle the bottleneck caused by the *cpu* action.

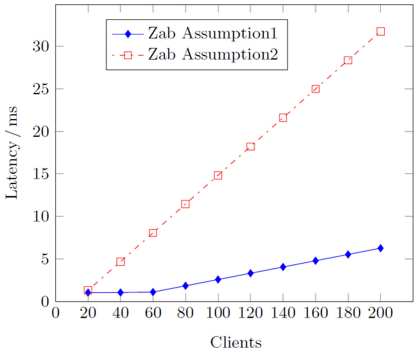

Fig. 4. Latency of Zab (*Assump.1* & *Assump.2*)

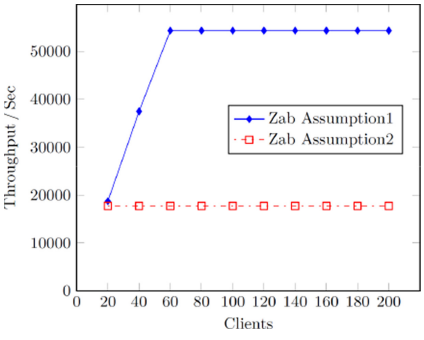

Fig. 5. Throughput of Zab (*Assump.1* & *Assump.2*)

6.3 Investigating Zab Assumption 2 Bottleneck

The system is overloaded quite rapidly by the use of *Assumption 2*, as depicted in the graphs of the Sect. 6.2. As a result, an additional research has been conducted

in order to solve the scalability issue of the Zab *Assumption 2*. In the model Zab *Assumption 2*, we have made an assumption that the action (*leaderDeliver*) will be used as a *cpu* action. Indeed, *leaderDeliver* is an action that occurs after the *commit*, where the *Leader* is proceeded to write physically on its memory. The *leaderDeliver* action is considered as one of the actions that extensively use the resources of the *Leader*. Consequently, it is renamed to *cpu* and its rate r_8 is used to calculate the average duration rate (c) of the *cpu* action, as illustrated in the Eq. 5. Due to the rate r_8 is the slowest rate among the other rates that used to calculate the rate of the *cpu*, we have varied the rate r_8 and observe how this will influence the behaviour of the Zab *Assumption 2* model.

Figure 6 and Fig. 7 are respectively displayed the latency and the throughput of the Zab *Assumption 2* model by means of varying the rate r_8. In Fig. 6 it is noticeable that the latency of the model is extremely decreased, with the increase of the rate r_8 value. Namely, it is declined from (31.74 ms) to (10.51 ms) when the active number of clients is equal to (200). The throughput also, has risen significantly in response to the change of the value of the r_8 rate as shown in Fig. 7. Whereas, the maximum reached throughput is (47588) request per second. Clearly, the model outcomes have illustrated that the overall performance is improved significantly by only increasing the rate r_8, which was limiting the *Leader* behaviour and causing the overload.

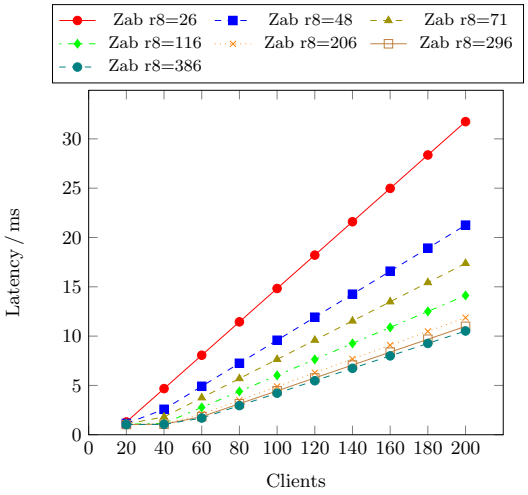

Fig. 6. Latency of Zab *Assump.2* varying (r_8)

6.4 Comparing Zab Model and Real Implementation of Zab

In this section, a comparison will be given between the performance of the Zab *Assumption 1* (2 *cpu*) model and a real implementation of Zab. This comparison aims to validate the model outcomes, specifically, the latency of the Zab

Assumption 1. Figure 8 showing the latency of both the Zab PEPA model and the implementation of Zab. As the load is relatively small at the beginning, it is noticeable that the latency of the Zab PEPA model is shown as a flat line, where the system has received (20, 40 and 60) requests. This is because there are enough resources and the coming requests are served without any waiting time.

However, the latency starts to increase linearly as the number of clients and requests are increased beyond (60), where the coming requests have to wait to be served. Although, the latency of the Zab model a bit higher (i.e. 6.25 ms to handle 200 requests) than the latency of the Zab implementation (i.e. 5 ms for the same number of requests); however, both are shown some consistency, in particular, the saturation point is the same for both, which is (60) clients. Then the latency of Zab model has risen linearly.

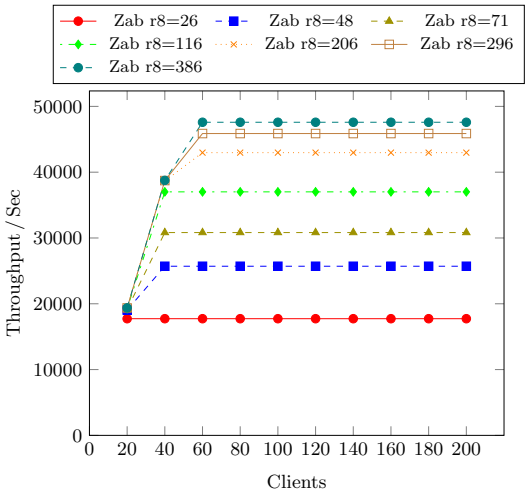

Fig. 7. Throughput of Zab *Assump.2* varying (r_8)

7 Related Work

High performance and scalability are desired requirements for cloud computing services providers and customers. A significant number of researches and studies have put forward this goal and not only presented practical solutions, but also, proposed designs of scalable systems. One example of using ZooKeeper coordination services is the study conducted by Skeirik *et al.* in [23], where they formally analysed a model of a ZooKeeper-based group key management service for fault-tolerance and performance. This formal analysis is based on rewriting logic model Maude [6] and the statistical model checking tool PVeStA [2]. This study also explored the reliability of ZooKeeper-based group key management

service for handling faults and providing a scalable performance with low latency. Although, the authors have considered security aspects of infrastructure in the cloud using formal modelling and they are shown that ZooKeeper is capable to tolerate with fault, but also authors stated that they have only considered one aspect of failure, i.e. crash-failures.

More recently, El-Sanosi and Ezhilchelvan [8] have presented an approach for improving the performance of ZooKeeper atomic broadcast protocol by coin tossing. This study presents a practical solution for the performance problem of a leader-based protocol; which raises with the increase of the load on the leader. A condition has been made that followers can broadcast only if the outcome of a coin toss is head, which reduces the communication traffic at the leader. However, there is a case where switching back to Zab would be the better choice, if one of followers fails or cannot compute the probability of coin toss. Nonetheless, significant performance improvement can be gained with the adoption of ZabCT proposed by [8] without impacting the performance of Zab.

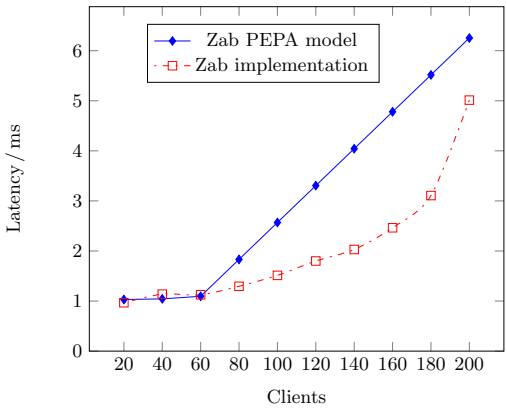

Fig. 8. Latency of Zab PEPA model (*Assump.1*) and Zab implementation

8 Conclusion

This research has presented an approach to the evaluation of the performance of the ZooKeeper atomic broadcast protocol, specifically the Zab broadcast phase. A Zab model has been created using PEPA Eclipse Plug-in tool, and an extensive performance analysis is conducted for the developed Zab model. This research is motivated by inspecting a high-performance coordination service system by the use of Performance Evaluation Process Algebra PEPA. In addition, validating the Zab PEPA model results against a real implementation of Zab.

At first, we have explored the scaling limit of the CTMC analysis, whereas the maximum number of clients that can be derived is (4), because of the well-known state space issue. On the other hand, the fluid flow analysis (ODEs) has

been used for a large scale system to derive two performance metrics (latency and throughput). Two versions of the Zab model with a slight difference (i.e. Zab *Assumption 1* and Zab *Assumption 2*) have been evaluated. The *Assumption 1* model has presented a high performance (low latency and high throughput). However, the system is overloaded with the use of *Assumption 2*. This is because of the *leaderDeliver* action, which is used as a *cpu* action in the Zab *Assumption 2* that has a relatively slow rate, hence, limited the behaviour of the Leader. Therefore, we have investigated further the scalability issue that shown in Fig. 4 and Fig. 5 of the model Zab *Assumption 2* by varying the rate r_8. Consequently, the performance of the model has improved significantly by increasing the rate r_8 as illustrated in Fig. 6 and Fig. 7, which is allowed to free up the Leader, thus performing much better for handling the increased loads.

Moreover, we have compared the latency of the model Zab *Assumption 1* with the latency of a real implementation of Zab, where the outcomes showing that they are noticeably consistent as depicted in Fig. 8; although the model gives a more idealised (smoother) performance due to the simplifications made.

In spite of the fact that we have based our analysis on the parameters that measured in a real system, but different real system can have slightly different parameters rates and that might affect the comparison between Zab *Assumption 1* and Zab *Assumption 2*, such that we may see better scalability on one than the other in our case. Furthermore, we have only looked at certain sets of parameters governed by a measurement, and also, we have only looked at one leader, if we have more leaders potentially can generate more capacity; but if there are coordinations between those leaders then it will limit that extra capacity. The model also accurately predicts the saturation point of the system (the point when throughput ceases to increase). Overall, the approach described in this study is highlighting a methodology to inspect a high-performance coordination services system, precisely, ZooKeeper (Zab protocol). Herein, we have only considered one phase of Zab (broadcast), therefore our future work is to model all phases of the Zab; taking into account further investigation in different performance aspects.

More recently, a new scalable distributed coordination service called Giraffe is released, which supposed to be an alternative to the coordination services, such as Zookeeper and Chubby. Shi *et al.* [22] claimed that Giraffe is 300% faster than Zookeeper. However, there is no independence analysis to support this study. So, analysing the performance of this system and compare it with the ZooKeeper would be an interesting point of research.

References

1. ZooKeeper, A.: https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index (2015). Accessed 16 Dec 2015
2. AlTurki, M., Meseguer, J.: PVeStA: a parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_28

3. Bernardo, M., Gorrieri, R.: A tutorial on EMPA: a theory of concurrent processes with nondeterminism, priorities, probabilities and time. Theoret. Comput. Sci. **202**, 1–54 (1998)
4. Borthakur, D., et al.: Apache Hadoop goes realtime at Facebook. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. SIGMOD 2011, pp. 1071–1080. ACM, New York, NY, USA (2011)
5. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 335–350. OSDI 2006, USENIX Association (2006)
6. Clavel, M., et al.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
7. Donatelli, S.: Superposed generalized stochastic petri nets: definition and efficient solution. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 258–277. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58152-9_15
8. EL-Sanosi, I., Ezhilchelvan, P.: Improving ZooKeeper atomic broadcast performance by coin tossing. In: Reinecke, P., Di Marco, A. (eds.) EPEW 2017. LNCS, vol. 10497, pp. 249–265. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66583-2_16
9. Fan, W., Bifet, A.: Mining big data: current status, and forecast to the future. SIGKDD Explor. Newsl. **14**(2), 1–5 (2013)
10. Harrison, P.G., Strulo, B.: Spades - a process algebra for discrete event simulation. J. Logic Comput. **10**(1), 3–42 (2000)
11. Hayden, R.A., Bradley, J.T.: Fluid-flow solutions in PEPA to the state space explosion problem. In: 6th Workshop on Process Algebra and Stochastically Timed Activities (PASTA), p. 25 (2007)
12. Hillston, J.: Fluid flow approximation of PEPA models. In: Second International Conference on the Quantitative Evaluation of Systems (QEST 2005), pp. 33–42 (2005)
13. Hillston, J.: A compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (2008)
14. Hillston, J., Gilmore, S.: Performance evaluation process algebra. http://www.dcs.ed.ac.uk/pepa/about/ (2011). Accessed 05 April 2016
15. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: USENIX Annual Technical Conference, vol. 8, p. 9 (2010)
16. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: high-performance broadcast for primary-backup systems. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN), pp. 245–256 (2011)
17. Junqueira, F., Reed, B.: ZooKeeper: Distributed Process Coordination. O'Reilly Media, Inc. (2013)
18. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
19. MacCormick, J., Murphy, N., Najork, M., Thekkath, C.A., Zhou, L.: Boxwood: abstractions as the foundation for storage infrastructure. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6. OSDI 2004, USENIX Association (2004)
20. Medeiros, A.: Zookeeper's atomic broadcast protocol: Theory and practice. Technical report (2012). Accessed 07 Oct 2015

21. Reed, B., Junqueira, F.P.: A simple totally ordered broadcast protocol. In: proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware. ACM (2008)
22. Shi, X., et al.: GIRAFFE: a scalable distributed coordination service for large-scale systems. In: 2014 IEEE International Conference on Cluster Computing (CLUSTER), pp. 38–47, September 2014
23. Skeirik, S., Bobba, R.B., Meseguer, J.: Formal analysis of fault-tolerant group key management using ZooKeeper. In: Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing 2013, pp. 636–641, May 2013
24. Tribastone, M., Duguid, A., Gilmore, S.: The PEPA Eclipse Plugin. SIGMETRICS Perform. Eval. Rev. **36**(4), 28–33 (2009)
25. Zimmerman, J.: Apache zookeeper in netflix. http://techblog.netflix.com/2011/11/introducing-curator-netflix-zookeeper.html (2011). Accessed 19 Jan 2016
26. Zookeeper, A.: Zookeeper: a distributed coordination service for distributed applications. https://zookeeper.apache.org/doc/trunk/zookeeperOver.html (2014). Accessed 16 Feb 2017