# Performance modelling and simulation of skewed demand in complex systems

Stephen Shephard

School of Computing Science, Newcastle University, Newcastle upon Tyne, NE1 7RU
`s.shephard2@newcastle.ac.uk`

**Abstract.** On-line Transaction Processing (OLTP) applications must frequently deal with the issue of skewed demand for some resources. This demand may overwhelm the whole system, affecting the owner's reputation and revenue. This paper presents system architectures for a ticketing use case using a selection of distributed computing technologies of the Cloud. It proposes models of these architectures and uses them to predict throughput in skewed demand scenarios. The experimental results of the models are then tested against simple built systems.

**Keywords:** Cloud, middleware, microservices, distributed databases, modelling, performance

## 1 Introduction

There are many high-profile examples of whole IT systems brought down by customer demand for part of their services. Customers were prevented from using any part of the London 2012 Olympic ticketing website on launch day to avoid demand overloading the system [22]. Demand for the finale of 'True Detective' [10] brought down HBO Go. Apple's iTunes Store suffered outage on the launch day of the iPhone 7 (new iPhone registration is carried out via an iTunes function) [28].

This paper claims that it is possible to design and build more resilient systems through effective use of Cloud technologies where higher than normal demand for one function or type of resource would not block access to the others. Skewed demand may be isolated so that it only affects parts of a system, or shared equally between different components. (The system may also adapt to demand by elastic scaling of resources, but this will not be considered as part of this paper). When combining a middleware solution with a distributed database, however, where is the system bottleneck? If there are levels of demand that cannot be met on a limited budget, and that therefore some components will no longer meet the required throughput, what is the impact on the remainder of the system?

The paper is organised as follows. Section 2 presents a use case based on ticket booking for a multi-sport event. Section 3 selects a set of distributed technologies, and section 4 discusses modelling in PEPA. Simple component models are proposed and tested in section 5 and composed into more complex system models that make end-to-end predictions. The models will then be tested against

actual systems in section 7, built using Java and the Java Spring framework [23], Cassandra [17][3] databases, and where appropriate Microsoft Azure Storage Queues [21]. These systems will be instrumented with Coda Hale Metrics [8] and measured under different scenarios of skewed demand, using Apache JMeter test plans [4]. Finally section 8 evaluates the approach and makes suggestions for future work.

### 1.1   Aims and Objectives

The aim is to demonstrate that PEPA (Performance Evaluation Process Algebra) [16] may be used to construct models that are good predictors of the behaviour of real systems under skewed demand scenarios, and provide insights into the effectiveness of Cloud technologies for handling skewed demand.

   The objectives are to:

- show that distributed databases and middleware queues may be modelled in PEPA.
- show that these PEPA component models may be used to test skewed demand.
- show that the component models may be composed into system models and their behaviour under test is different to the components alone.
- build test system architectures for the PEPA models and show that the test system behaviour matches that predicted by the models.
- identify areas of further work.

## 2   Background

Consider a general OLTP application using a distributed architecture, as shown in Figure 2. Users access the application with a web-based front end. Resources are stored in one or more databases. In between the web servers and database are worker applications that service user requests, connected to the web servers by some middleware. There are strategies for coping with skewed demand at each layer of this architecture.

*Adapting.* A system using *elastic scaling* may adapt to increased demand. Rapid elasticity is an essential characteristic of Cloud Computing by the NIST definition [20]. Computing resources, for example web servers or worker applications, can be elastically and often automatically scaled to meet current demand. This gives the appearance of resources that are limited only by the system owner's budget.

*Sharing.* High demand may be shared between resources. HTTP load balancing improves the scalability of a web-based application by distributing the demand across multiple web servers [13]. Shared middleware such as a point-to-point queue, provides a competing consumer pattern to balance load from several producers, e.g. web servers, between multiple consumers e.g. worker applications.
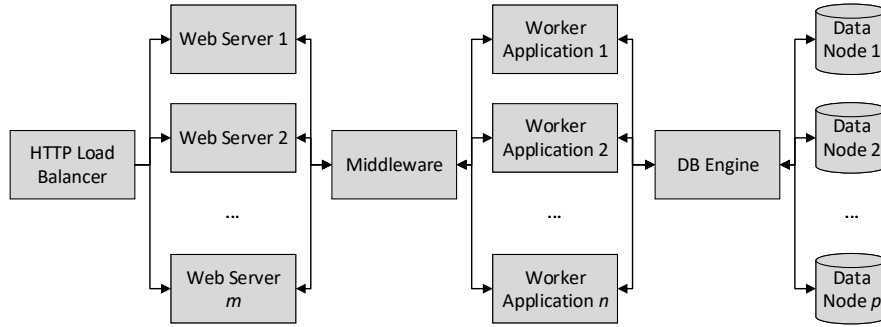
**Fig. 1.** OLTP application distributed architecture

*Isolating.* If it is not possible to satisfy the skewed demand within a given budget, then it may be appropriate to isolate that demand from the rest of the system. Horizontal partitioning of a distributed database can place high demand resources on different data nodes. Microservices architecture offers a pattern for partitioning the data resources, the worker applications and the web servers, connecting them into entirely separate smaller end-to-end services.

### 2.1   Use Case

The concrete use case for constructing models and building systems is a ticketing application. Following the Olympic example given in the Introduction, tickets will be for a multi-sport event. Some sports are more popular than others and it will be assumed that there will be predictable skewed demand for *athletics* tickets. Use cases where the resources with skewed demand are unknown - where they are only discovered once the application goes online, requiring some adaptive approach - are out of scope.

The application has three possible operations:

1. Search (for available tickets)
2. Book (allocate a ticket to a customer)
3. Return (customer releases a ticket allocation)

Such a ticketing application may be generalised to any system for allocating and releasing other resources with variable demand.

This paper considers the problem of higher than average demand for a particular type of ticket, and to what extent the system will allow users to search for other ticket types if some component is overloaded by the skewed demand for the most popular tickets. It does not consider issues of fair allocation of scarce resources.

## 3   Technologies

### 3.1   Scope

**In Scope.** The selected technologies in scope of this paper are shared middleware queues, distributed databases and microservices, which are discussed in more detail below. The database partitioning strategy and entirely separate databases for the microservices architecture offer alternative means of isolating the skewed demand. Using a single middleware queue shares and distributes the demand, this time in contrast to the microservices middleware approach. The models will compare these approaches and investigate the behaviour of systems where the components have conflicting approaches to handling demand.

**Out of Scope.** The paper will not consider elastic scaling or HTTP load balancing. There is already a great deal of work in evaluating right-sizing strategies (minimising underutilisation and overutilisation of compute resources) for the former, e.g. [1], [11], [19]. HTTP load balancing is a relatively mature technology, and work has been done on simulation to evaluate different algorithms by response time and web server utilisation [5].

### 3.2   Queue Middleware

Good choice of middleware in a system will help ensure that its components are connected, but loosely coupled. If, for example, a web server is blocked waiting for a response from a worker application carrying out a more expensive operation, then the throughput of the web server will be limited to that of the worker application. The use case 'return' operation however does not require a direct response from the system. As long as the customer can rely on eventual guaranteed delivery of the return request, (and that the cost of their ticket will be refunded) then they do not need to wait for a direct response to their return.

Point-to-Point Queues, e.g. Azure Storage Queues [21], are a form of Message-Oriented Middleware - an asynchronous, brokered message service providing an intermediate layer between senders and receivers, decoupling their communication. Message delivery may take minutes rather than milliseconds, but the service providers do provide configurable delivery guarantees [9]. With synchronous middleware such as Remote Procedure Call (RPC), the calling process is blocked until the called service completes and returns control to the caller. Distributed systems using asynchronous middleware do not block when calling a remote service. Control is immediately passed back to the caller, and a response may be returned eventually, with the caller polling the remote service for the response, or the remote process calling a method in the caller to send the response.

Many processes may send messages to a queue, and each message is received by one consumer - though it may be one of several consumers competing for messages from this queue. This competing consumer pattern offers a means of balancing load from the Web Servers between the Worker Applications in the ticketing use case.

### 3.3   Microservices

Microservices architecture is an approach to structuring applications as suites of small services, defined by business capability verticals rather than technological layers [18] [24]. Each of the use case requirements - search for, book or return tickets - might typically be microservices with their own worker applications and data nodes. Ticket data would be denormalised across the data nodes and made eventually consistent via a backplane messaging service [25] as shown in Figure 3.3. This would certainly isolate the demand for search, book and return from each other - returning tickets would not be blocked by a system where bookings were overloaded. In the ticketing use case however, there is skewed demand for athletics tickets. In a real-world system the booking microservice might be further broken down to a lower level of granularity to deal with this, i.e. a separate microservice for booking each ticket type.
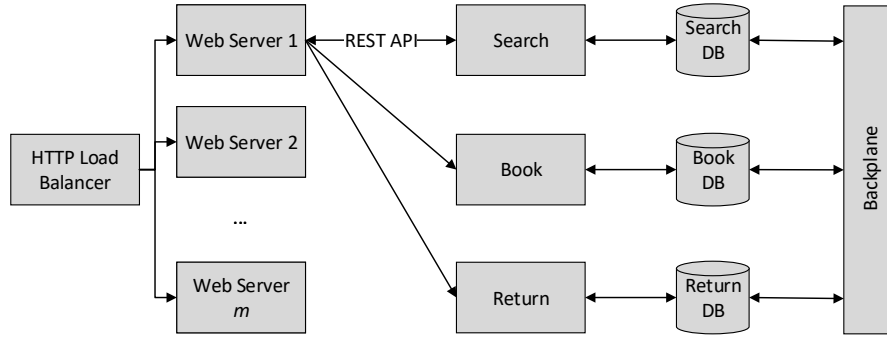


**Fig. 2.** Microservices

### 3.4   Distributed databases

Modern SQL and NoSQL databases are designed to scale both data and the load of operations accessing that data over many servers that do not share disk or RAM, so-called 'shared nothing' architecture [7]. We may partition data *vertically*, dividing tables into groups of columns that may be placed on different data nodes; or *horizontally*, where the split is by row [2].

In the use case, the quantity of data does not approach the levels of 'Big Data' applications. Partitioning is proposed instead as a means of scaling the demand for that data. The ticketing system will not require a large number of columns and the three operations outlined do not have significantly different column requirements, therefore horizontal partitioning is most relevant. The partition key of a Ticket table may be the Ticket Type, the Date, or the seat Row. Demand

for tickets is likely to vary by each of these attributes. An alternative partitioning strategy would be to on denormalised tables supporting the query, book and return operations. The load on each data node would follow the demand for the data types and operations placed there.

One issue to be aware of is *replication*. Most distributed databases offer replication of data from one partition to another for availability. In the use case, if demand overloads a data node, the database may share the throughput using a copy of the data on another data node. If this is also the primary data node of an otherwise low demand data type, then it may be overwhelmed in turn i.e. the skewed demand has followed the data.
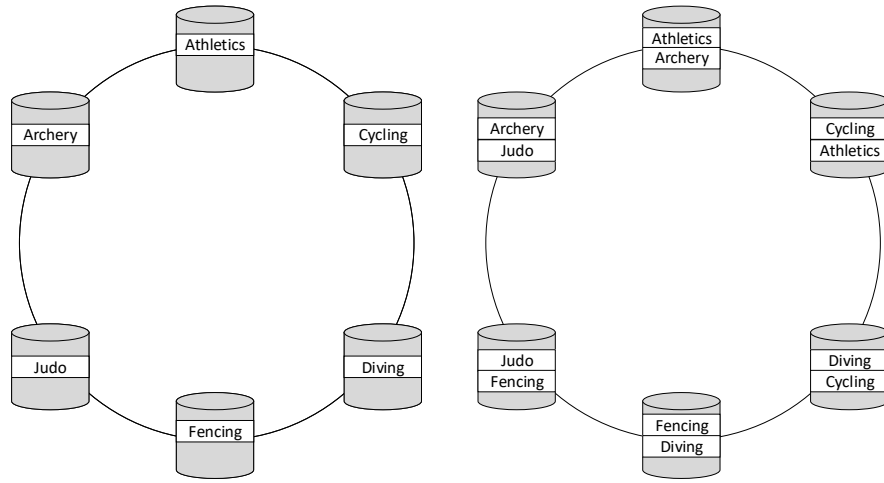


**Fig. 3.** Consistent hashing, without and with replication

The Cassandra database has an interesting method of partitioning data, using *consistent hashing* (also used by Riak, Redis and BigData among others [15]). The largest output of a hash function wraps round to the smallest value so that the range of hash values forms a conceptual 'ring'. Each data node is assigned a position on this ring, then the hash value of the partition key of a data item is used to determine the node used to store it. When using replication, with a replication factor of *N*, a copy of the data is placed on the next *N-1* nodes walking clockwise round the ring [17]. This is illustrated in Figure 3.4.

## 4 Modelling

The modelling technique must enable predictions about throughput for varying levels of skewed demand. It must also be possible to compose system models from simpler components. Two approaches for the latter are programming or mathematical language-based models (e.g. *CloudSim* or *Process Algebra* respectively).

*CloudSim.* CloudSim [6] is a Java framework for developing cloud datacentre simulations. Much of it is concerned with modelling the efficient running of that infrastructure, for example the power usage, but it also includes utilisation models and may be useful for predicting the effect of elastic scaling.

CloudSim simulations require Java development for creation and modification, which is an overhead in building the models but offers flexibility in applying them.

*Process Algebra.* Process Algebras (such as PEPA or TIPP [14]) model throughput in interdependent processes, with a mixture of independent and shared actions operating at different rates. There is a PEPA Eclipse plugin [30] that allows PEPA specifications to be parsed and run like programs, aiding experimentation on a range of action rates by automating repetitive calculations.

### 4.1 PEPA (Performance Evaluation Process Algebra)

The models will be produced using PEPA. This paper is concerned with distribution of throughput in complex systems, rather than right-sizing those systems. The PEPA Workbench will allow the automation of testing with a range of skewed demand values.

A PEPA model describes a system of interacting *components* which carry out *activities* at specified or passive *rates*. A component is usually denoted by a name with an initial upper case letter, e.g. *Website*, and an activity type and rate are expressed as a bracketed pair e.g. $(request, r)$ where the activity type is normally a full lower case name and the rate is a single letter or the top symbol $\top$, denoting an unspecified (passive) rate. There is a set of combinators that describe how the components and activities interact. This paper uses the following subset, for the full syntax see [16]:

**Prefix:** $(\alpha, r).P$ - a component carries out activity $\alpha$ at rate $r$ and then behaves as component $P$.

**Constant:** $A \stackrel{def}{=} P$ - assign the behaviour of component $P$ to the constant $A$. Used with prefix, this can be used to define a recurring process e.g. $P \stackrel{def}{=} (\alpha, r).P$.

**Choice:** $P + Q$ - a component may behave *either* as component $P$ or $Q$, non-deterministically. This represents a race condition between components.

**Cooperation:** $P \bowtie_L Q$ - for shared activities in the set $L$, components $P$ and $Q$ may only proceed with the simultaneous execution of those activities at the rate of the slowest component, otherwise they behave independently.

**Parallel:** $P \parallel Q$ - shorthand for components that synchronize with no shared activities i.e. equivalent to $P \bowtie_\emptyset Q$.

**Aggregation:** $P[N]$ - represents $N$ instances of component $P$, but does not distinguish which instance of $P$ changes. So for example where some component has states $P1$ and $P2$, and $(P1|P2)$ does not equal $(P2|P1)$, this model has 4 states. If it doesn't matter which component has changed, then the model has only 3 states and can be written as $P[2]$.

*Solutions.* PEPA models are used to represent a system using a stochastic process, where the activity durations are random variables. Where these are negative exponentially distributed, then this representation is a continuous time Markov process with a steady state solution over a period of time [16]. This may be calculated using the PEPA Eclipse plugin.

## 5    PEPA Component Models

The first stage is to create suitable PEPA models for the selected technology components from section 3, simple enough to be composed into more complex system models but still able to demonstrate interesting behaviour. These models are tested using the PEPA Eclipse plugin [30] to calculate the steady-state throughputs of each activity for a given range of input rates for the activity with skewed demand. The results are analysed to verify that the component models behave as expected for the technologies, and to discover any additional insights. Note that there is no microservices component model. Microservices is an architecture and will be shown in the System Modelling section.

### 5.1    Shared middleware queue

Work has already been done on modelling queueing systems in PEPA [29]. A single queue with a limited buffer size of $N$ may be written as (service and arrival components not shown, for brevity):

$$Queue_0 \stackrel{def}{=} (arrival, \top).Queue_1$$
$$Queue_j \stackrel{def}{=} (arrival, \top).Queue_{j+1} + (service, \top).Queue_{j-1}, 1 \leq j \leq N - 1$$
$$Queue_N \stackrel{def}{=} (service, \top).Queue_{N-1}$$

Using aggregation, this may be more simply represented in an easily extensible form as $Queue[N]$. The limitation of this representation however is that it makes no distinction between the states of individual queue position instances, only the numbers of instances in each state. This means there is no ordering guarantee e.g. the First In First Out (FIFO). Actual cloud service queues do not necessarily implement FIFO, for example Azure Storage Queues [21] do not guarantee it.

For the skewed demand use case, a queue must be able to support arrival actions at different rates, and must potentially be able to support service actions in different ways too. Again [29] suggests an approach for this model, with a queue synchronised with a linear combination of components with different characteristics. Thus the PEPA model for a general shared queue is shown in Figure 5.1.

There are two arrival processes, one dealing with the arrival of cycling requests at a uniform, normal rate (here set to $c = 1.0$) and one dealing with the athletics requests at skewed rates (starting at the same rate as cycling requests, and increasing in steps of 1.0 to 10.0). The queue itself is an aggregation of $N$ components each of which has three states; an empty instance $Q_0$, 'filled' with an athletics request $Q_A$, or filled with a cycling request $Q_C$. Finally there are also two service processes, generalised here to $Service_1$ and $Service_2$ (in some of the system models, they will not necessarily be dedicated to serving athletics or cycling requests), both with the same maximum service rate of $s = 5$. The higher rates of skewed demand are therefore more than the service processes can handle.

$$a = 1.0 - 10.0$$
$$c = 1.0$$
$$s1 = 5.0$$
$$s2 = 5.0$$

$$Arrival_A \stackrel{def}{=} (athletics, a).Arrival_A$$
$$Arrival_C \stackrel{def}{=} (cycling, c).Arrival_C$$
$$Service_1 \stackrel{def}{=} (serve1, s1).Service_1$$
$$Service_2 \stackrel{def}{=} (serve2, s2).Service_2$$

$$Q_0 \stackrel{def}{=} (athletics, \top).Q_A + (cycling, \top).Q_C$$
$$Q_A \stackrel{def}{=} (serve1, \top).Q_0$$
$$Q_C \stackrel{def}{=} (serve2, \top).Q_0$$

$$Arrival_A \underset{athletics}{\bowtie} Q_0[N] \underset{serve1}{\bowtie} Service_1 \underset{cycling}{\bowtie} Arrival_C \underset{serve2}{\bowtie} Service_2$$

**Fig. 4.** Shared queue PEPA model

The model is tested in the Eclipse plugin using a series of different queue lengths $N$ and for different rates of athletics demand $a$ from 1 to 10. This provides the actual throughputs in steady state of each activity. Figure 5.1 shows the throughput of *athletics* for exponentially increasing queue lengths from 1 to 20. Figure 5.1 shows the same for *cycling*. Table 1 shows the numerical results for a queue of length N=10. The results demonstrate that:

- the throughput of *athletics* is constrained by the maximum service rate of the process handling those requests.
- the throughput of *cycling* is constrained by the ratio between the input rates of athletics and cycling. When the input rate of athletics requests is 10 times that of cycling requests, then the queue holds these requests in a 10:1 ratio. As the actual *athletics* throughput may not exceed the service rate $s = 5$, then the cycling throughput is throttled to 0.5.
- for larger queue sizes, the throughputs approach their maximum limits. A real queue service has an effectively unlimited length, but in PEPA models the state space quickly becomes too large for the Eclipse plugin to handle. It is a useful result, therefore, to find that for a maximum skewed demand of 10 times the normal demand, a queue length of 10 gives a practical and well performing model.
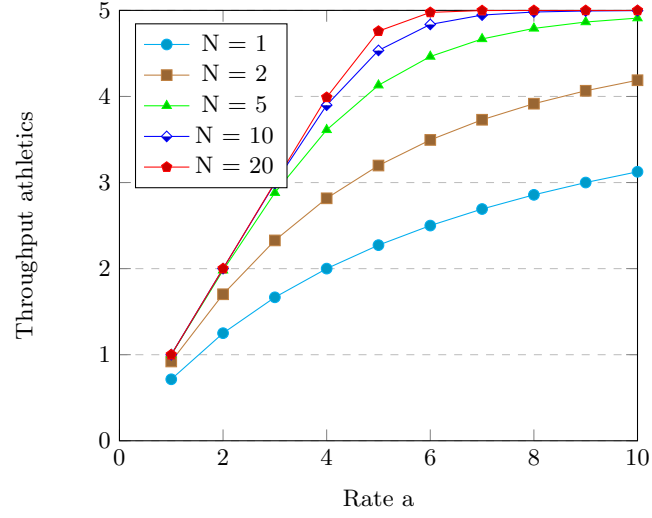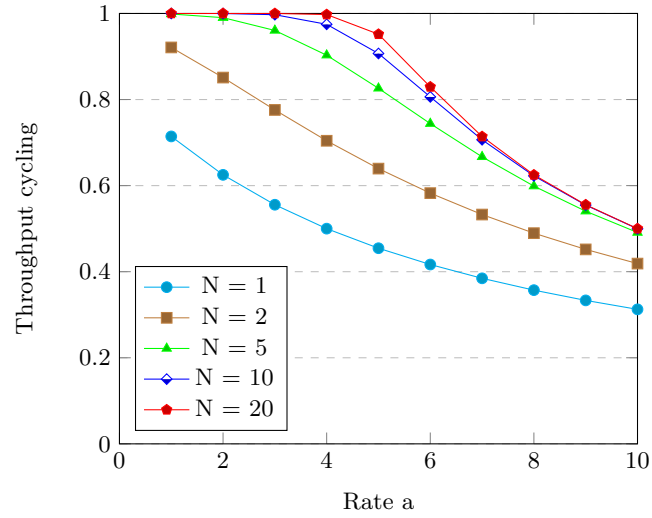
Throughput of athletics against input rate a for different queue lengths N



**Fig. 5.** Shared queue experimental results - athletics

Throughput of cycling against input rate a for different queue lengths N



**Fig. 6.** Shared queue experimental results - cycling

**Table 1.** Shared queue N=10 experimental results

| Rate | Throughput | | | | |
|------|------------|---------|-------|--------|--------|
| a    | athletics  | cycling | ratio | serve1 | serve2 |
| 1    | 1          | 1       | 1     | 1      | 1      |
| 2    | 2          | 1       | 2     | 2      | 1      |
| 3    | 2.99       | 1       | 3     | 2.99   | 1      |
| 4    | 3.9        | 0.97    | 4     | 3.9    | 0.97   |
| 5    | 4.53       | 0.91    | 5     | 4.53   | 0.91   |
| 6    | 4.84       | 0.81    | 6     | 4.84   | 0.81   |
| 7    | 4.95       | 0.71    | 7     | 4.95   | 0.71   |
| 8    | 4.98       | 0.62    | 8     | 4.98   | 0.62   |
| 9    | 4.99       | 0.55    | 9     | 4.99   | 0.55   |
| 10   | 5          | 0.5     | 10    | 5      | 0.5    |

## 5.2   Database models

A very simple representation of a single database process is a component that receives a request for data (either read or write) at some rate based on demand, and serves it at a rate based on the database's performance:

$$DB \stackrel{def}{=} (request, r).(dbsrv, db).DB$$

This is a highly abstract representation. It does not model features such as session management, parallelism, caching, locking or any notion that data manipulation statements vary in complexity and expense. Nevertheless it is a useful building block for distributed databases, as shown below.

**Distributed database.** Figure 5.2 shows a model of a distributed database, where the data has been partitioned by sport onto two different database nodes with identical performance. The data request activities are *athletics* and *cycling*. These may represent search, book or return operations on athletics or cycling tickets. Users may search for either type of ticket from the website component, and the code or database engine will route the search to the correct data node. Thus $DB_1$ here is able to service *athletics* requests, at a maximum rate of *db*, and $DB_2$ can service *cycling* requests at the same rate (the model assumes homogeneous database nodes to reduce the variables under consideration, although as there are separate database service processes it is extensible to heterogeneous systems). Both nodes execute in parallel without cooperating on any activities.

$$a = 1.0 - 10.0$$
$$c = 1.0$$
$$db = 5.0$$

$$Website \stackrel{def}{=} (athletics, a).\,Website + (cycling, c).\,Website$$
$$DB_1 \stackrel{def}{=} (athletics, \top).DBsrv_1$$
$$DBsrv_1 \stackrel{def}{=} (dbsrv1, \top).DB_1$$
$$DB_2 \stackrel{def}{=} (cycling, \top).DBsrv_2$$
$$DBsrv_2 \stackrel{def}{=} (dbsrv2, \top).DB_2$$
$$Service_1 \stackrel{def}{=} (dbsrv1, db).Service_1$$
$$Service_2 \stackrel{def}{=} (dbsrv2, db).Service_2$$

$$Website \underset{\substack{athletics \\ cycling}}{\bowtie} DB_1 \parallel DB_2 \underset{\substack{dbsrv1 \\ dbsrv2}}{\bowtie} Service_1 \parallel Service_2$$

**Fig. 7.** Distributed database PEPA model

Experiments are carried out in the Eclipse plugin by fixing the input rate of $db$ at 5.0, the rate $c$ of cycling requests to 1.0 and by testing each input rate $a$ of athletics requests from 1.0 to 10.0 in steps of 1.0, to simulate increasing levels of skewed demand for athletics tickets which becomes too high for a single database node to service. Table 2 shows the numerical results, Figure 5.2 shows the throughput of both *athletics* and *cycling* against the skewed input rate $a$, and Figure 5.2 shows the throughput of each database node against the same range of inputs. The results show that:

- the throughput of *athletics* is constrained by the maximum service rate of the database node handling those requests, and both athletics and cycling activities demonstrate loss of throughput.
- the throughput of *cycling* is independent of athletics.
- the database node throughput follows the throughput of each sport activity, i.e. the partitioning strategy routes all the demand onto the node handling that sport.
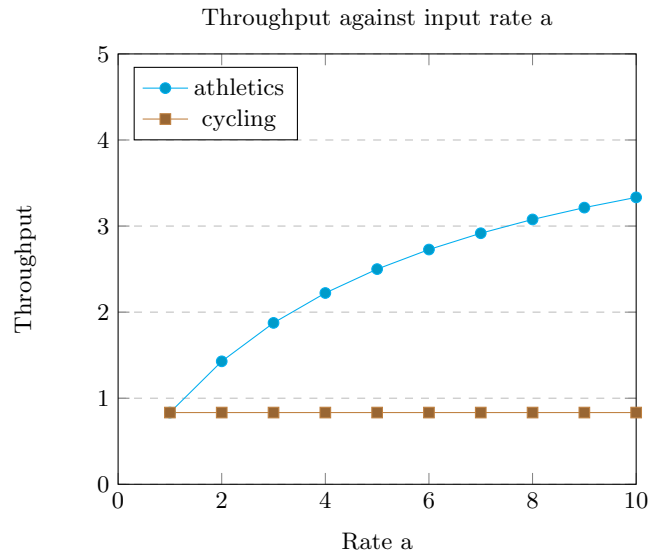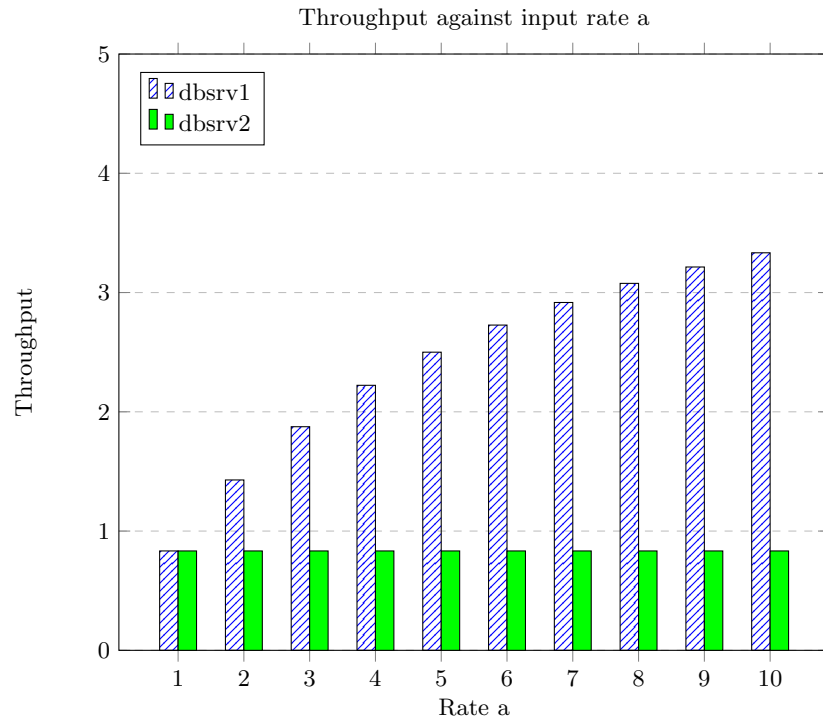
Throughput against input rate a



**Fig. 8.** Distributed database without replication - sport throughput

Throughput against input rate a



**Fig. 9.** Distributed database without replication - database throughput

**Table 2.** Distributed database without replication experimental results

| Rate | Throughput | | | |
|---|---|---|---|---|
| a | athletics | cycling | dbsrv1 | dbsrv2 |
| 1 | 0.83 | 0.83 | 0.83 | 0.83 |
| 2 | 1.43 | 0.83 | 1.43 | 0.83 |
| 3 | 1.88 | 0.83 | 1.88 | 0.83 |
| 4 | 2.22 | 0.83 | 2.22 | 0.83 |
| 5 | 2.5 | 0.83 | 2.5 | 0.83 |
| 6 | 2.73 | 0.83 | 2.73 | 0.83 |
| 7 | 2.92 | 0.83 | 2.92 | 0.83 |
| 8 | 3.08 | 0.83 | 3.08 | 0.83 |
| 9 | 3.21 | 0.83 | 3.21 | 0.83 |
| 10 | 3.33 | 0.83 | 3.33 | 0.83 |

**Distributed database with replication.** Constructing and meaningfully testing a model of a distributed database using consistent hashing with replication requires at least three types of sport tickets, so a *diving* activity is introduced. The partitioning strategy is as the previous model, but now $DB_1$ is able to service *athletics* and *cycling* requests, $DB_2$ handles *cycling* and *diving*, and $DB_3$ handles *athletics* and *diving* as shown in Figure 5.2. This model makes a key assumption that each data node handles each supported sport with equal probability, and once again each node operates at an identical rate that is insufficient to meet the skewed demand.

Experiments are carried out in the Eclipse plugin by fixing the input rate of *db* at 5.0, the rates *c* and *d* of cycling and diving requests to 1.0 and by testing each input rate *a* of athletics requests from 1.0 to 10.0 in steps of 1.0. The numerical results in Table 3 show that the throughputs of cycling and diving are identical. Figure 5.2 therefore shows only the throughput of *athletics* and *cycling* against the skewed input rate *a* (the plots of cycling and diving would be superimposed). Figure 5.2 shows the throughput of all three database nodes against the same range of inputs. The results show that:
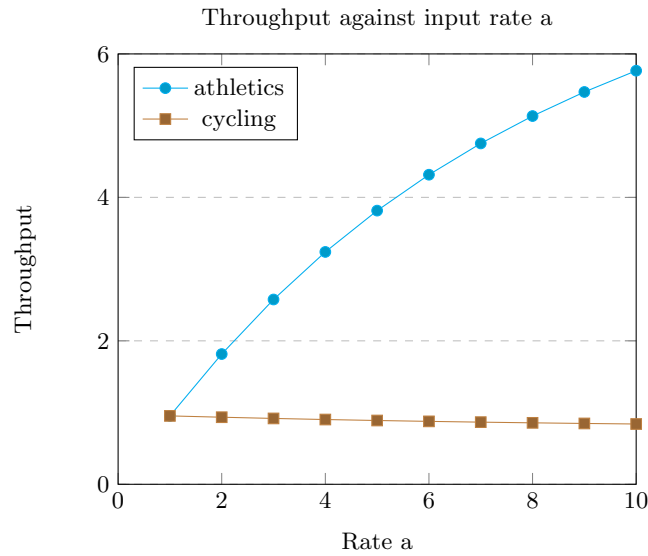
- the throughput of *athletics* is higher than for the distributed database without replication. The extra demand has been shared between both data nodes supporting athletics requests.
- the throughput of *cycling* and *diving* are no longer independent of athletics. They no longer reside only on data nodes that are independent of athletics demand.
- the database node throughput of the two nodes supporting athletics requests is equal, and both are higher than the remaining data node that supports only cycling and diving. The throughput of this node increases as athletics throughput increases however, suggesting the node is picking up an increasing proportion of the constant demand for cycling and diving tickets.

$$a = 1.0 - 10.0$$
$$c = 1.0$$
$$d = 1.0$$
$$db = 5.0$$

$$Website \stackrel{def}{=} (athletics, a).Website + (cycling, c).Website + (diving, d).Website$$
$$DB_1 \stackrel{def}{=} (athletics, \top).DBsrv_1 + (cycling, \top).DBsrv_1$$
$$DBsrv_1 \stackrel{def}{=} (dbsrv1, \top).DB_1$$
$$DB_2 \stackrel{def}{=} (cycling, \top).DBsrv_2 + (diving, \top).DBsrv_2$$
$$DBsrv_2 \stackrel{def}{=} (dbsrv2, \top).DB_2$$
$$DB_3 \stackrel{def}{=} (diving, \top).DBsrv_3 + (athletics, \top).DBsrv_3$$
$$DBsrv_3 \stackrel{def}{=} (dbsrv3, \top).DB_3$$
$$Service_1 \stackrel{def}{=} (dbsrv1, db).Service_1$$
$$Service_2 \stackrel{def}{=} (dbsrv2, db).Service_2$$
$$Service_3 \stackrel{def}{=} (dbsrv3, db).Service_3$$

$$Website \underset{\substack{cycling \\ diving}}{\overset{\substack{athletics}}{\bowtie}} DB_1 \parallel DB_2 \parallel DB_3 \underset{\substack{dbsrv2 \\ dbsrv3}}{\overset{\substack{dbsrv1}}{\bowtie}} Service_1 \parallel Service_2 \parallel Service_3$$

**Fig. 10.** Distributed database with replication PEPA model

**Table 3.** Distributed database with replication experimental results

| Rate | Throughput | | | | | |
|---|---|---|---|---|---|---|
| a | athletics | cycling | diving | dbsrv1 | dbsrv2 | dbsrv3 |
| 1 | 0.95 | 0.95 | 0.95 | 0.95 | 0.95 | 0.95 |
| 2 | 1.82 | 0.94 | 0.94 | 1.34 | 1 | 1.34 |
| 3 | 2.58 | 0.92 | 0.92 | 1.68 | 1.05 | 1.68 |
| 4 | 3.24 | 0.9 | 0.9 | 1.98 | 1.08 | 1.98 |
| 5 | 3.82 | 0.89 | 0.89 | 2.24 | 1.11 | 2.24 |
| 6 | 4.32 | 0.88 | 0.88 | 2.46 | 1.14 | 2.46 |
| 7 | 4.75 | 0.87 | 0.87 | 2.66 | 1.17 | 2.66 |
| 8 | 5.13 | 0.86 | 0.86 | 2.83 | 1.19 | 2.83 |
| 9 | 5.47 | 0.85 | 0.85 | 2.98 | 1.2 | 2.98 |
| 10 | 5.77 | 0.84 | 0.84 | 3.11 | 1.22 | 3.11 |

Throughput against input rate a



**Fig. 11.** Distributed database with replication - sport throughput

Throughput against input rate a
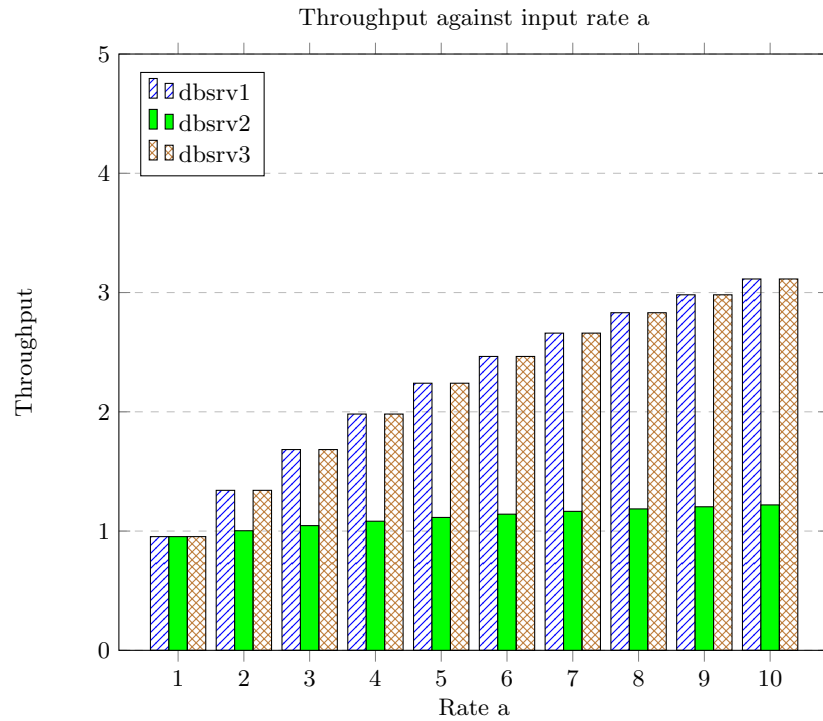


**Fig. 12.** Distributed database with replication - database throughput

## 6   PEPA System Models

The components are combined into models of full distributed architectures, that may be implemented and tested as working built systems. There are three system models - a simplified microservices architecture, and two models composed of a shared queue and distributed database, with and without replication.

### 6.1   Simple microservices

The simple microservices model has separate end-to-end services for handling athletics and cycling ticket requests. This is not a 'natural' microservices implementation, which would be more likely to separate on operations e.g. searching, booking and returning tickets. Choosing this design however makes the system directly comparable to the partitioning strategy used for the distributed database models. It is not in itself functionally different to separating services by operations unless considering additional features to handle eventual data consistency (see Future Work).

The system has two separate databases, one for athletics tickets, one for cycling, and each has its own dedicated worker application (Figure 6.1). The PEPA model of this (Figure 6.1) uses the same database process building blocks as the distributed database component models, but in this case they cooperate with the dedicated worker application processes $Worker_A$ and $Worker_C$.

As for the component models, there are two arrival processes, dealing with cycling requests at the rate $c = 1.0$ and athletics requests at rates 1.0 to 10.0 in steps of 1.0. Again each database may serve requests at a maximum rate of $db$. Note that this rate has been changed to 6.5 so that it is proportional to the performance observed when testing the built system (the model has been tuned following measurement of the system).

The worker application processes have a maximum rate of $w = 100.0$. This value has been chosen to be much higher than the other parts of the system to minimise its impact on the system testing. The assumption being made here is that the applications may be designed to cope with this much higher demand, perhaps using Elastic Scaling of servers, which is out of scope of this paper.

The PEPA Eclipse plugin experiments test each input rate $a$ of athletics requests from 1.0 to 10.0 in steps of 1.0, with all other rates fixed. Table 4 shows the numerical results, and Figure 6.1 shows the throughput of *athletics* and *cycling* against input rate $a$. These demonstrate that:

  - the throughput of *athletics* is constrained by the maximum service rate of the database handling those requests. Both athletics and cycling activities demonstrate some loss of throughput, though less than with the distributed database component (perhaps due to the additional worker application processes producing a partial decoupling effect).
  - the throughput of *cycling* is independent of athletics. This supports the claim that microservices architecture isolates the skewed demand.
  - the database node throughput follows the throughput of each sport activity.
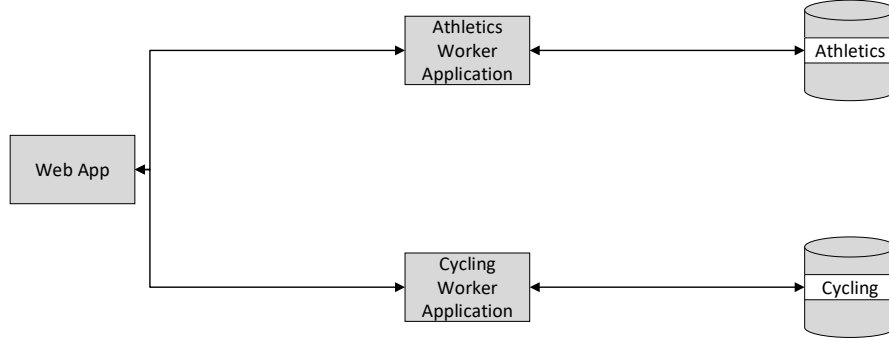
**Fig. 13.** Simple microservices architecture

$$a = 1.0 - 10.0$$
$$c = 1.0$$
$$w = 100.0$$
$$db = 6.5$$

$$Website \stackrel{def}{=} (athletics, a).Website + (cycling, c).Website$$
$$Worker_A \stackrel{def}{=} (athletics, \top).WorkerSrv_A$$
$$WorkerSrv_A \stackrel{def}{=} (workerA, \top).Worker_A$$
$$Worker_C \stackrel{def}{=} (cycling, \top).WorkerSrv_C$$
$$WorkerSrv_C \stackrel{def}{=} (workerC, \top).Worker_C$$
$$DB_1 \stackrel{def}{=} (workerA, w).DBsrv_1$$
$$DBsrv_1 \stackrel{def}{=} (dbsrv1, db).DB_1$$
$$DB_2 \stackrel{def}{=} (workerC, w).DBsrv_2$$
$$DBsrv_2 \stackrel{def}{=} (dbsrv2, db).DB_2$$
$$Service_1 \stackrel{def}{=} (dbsrv1, db).Service_1$$
$$Service_2 \stackrel{def}{=} (dbsrv2, db).Service_2$$

$$Service_1 \underset{dbsrv1}{\bowtie} DB_1 \underset{workerA}{\bowtie} Worker_A \underset{athletics}{\bowtie} Website \underset{cycling}{\bowtie} Worker_C \underset{workerC}{\bowtie} DB_2 \underset{dbsrv2}{\bowtie} Service_2$$

**Fig. 14.** Simple microservices PEPA model

**Table 4.** Simple microservices experimental results

| Rate | Throughput | | | | | |
|---|---|---|---|---|---|---|
| a | athletics | cycling | workerA | workerC | dbsrv1 | dbsrv2 |
| 1 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 | 0.97 |
| 2 | 1.83 | 0.97 | 1.83 | 0.97 | 1.83 | 0.97 |
| 3 | 2.55 | 0.97 | 2.55 | 0.97 | 2.55 | 0.97 |
| 4 | 3.14 | 0.97 | 3.14 | 0.97 | 3.14 | 0.97 |
| 5 | 3.61 | 0.97 | 3.61 | 0.97 | 3.61 | 0.97 |
| 6 | 3.99 | 0.97 | 3.99 | 0.97 | 3.99 | 0.97 |
| 7 | 4.3 | 0.97 | 4.3 | 0.97 | 4.3 | 0.97 |
| 8 | 4.55 | 0.97 | 4.55 | 0.97 | 4.55 | 0.97 |
| 9 | 4.75 | 0.97 | 4.75 | 0.97 | 4.75 | 0.97 |
| 10 | 4.92 | 0.97 | 4.92 | 0.97 | 4.92 | 0.97 |
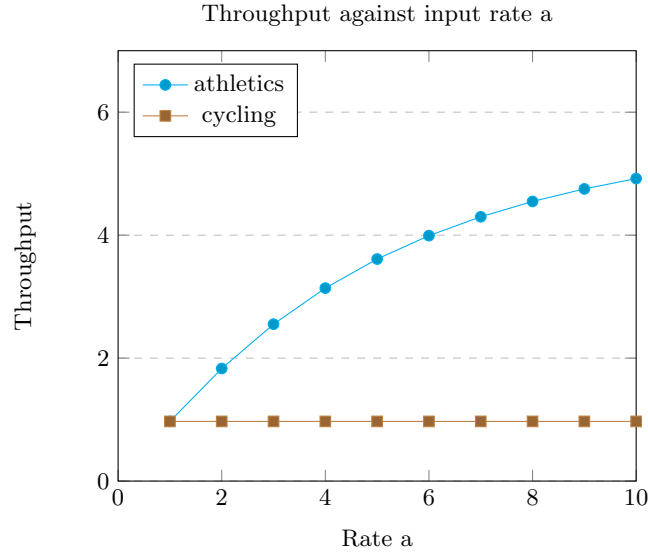


**Fig. 15.** Simple microservices experimental results

## 6.2    Shared queue and distributed database

The next system model (Figure 6.2) is the first combination of the shared queue and distributed database components. A website sends all ticket requests asynchronously to a cloud queue service, and one or more worker applications (either a multi-threaded application or a scaling set of applications) dequeues the requests and forwards them to the distributed database. The database uses a horizontal partitioning strategy based on the sport, without any replication.

The PEPA model (Figure 6.2) is a straightforward combination of the shared queue and distributed database component models. A queue length of N=10 is used as the experiments showed that for a small state space, this allowed the athletics throughput to get very close to the maximum service rate.

There is no separate representation of worker processes (the previous system model showed that the throughputs of the worker activities were exactly the same as the processes on either side of them) but as before a high maximum rate has specified $q = 100.0$ for the rate at which requests may be dequeued. This is the rate used for *queueA* and *queueB* requests arriving at the database processes.
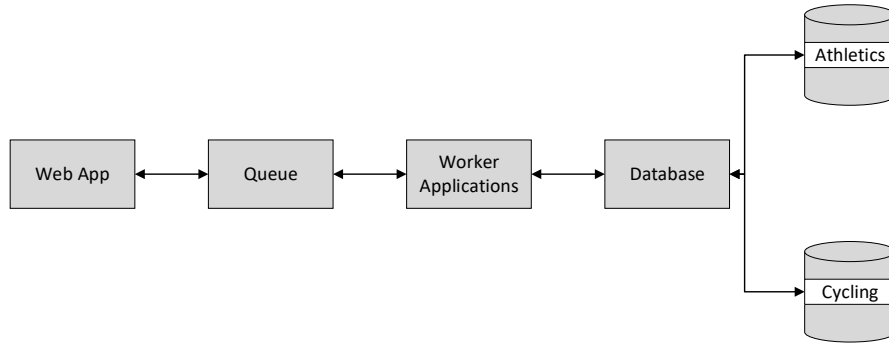


**Fig. 16.** Shared queue middleware architecture

As is now usual, the Eclipse plugin is used to find the steady-state throughputs of the activities for input rates of athletics requests increasing from 1.0 to 10.0 with cycling and other rates fixed. The results appear numerically in Table 5 and as a chart comparing *athletics* and *cycling* throughput in Figure 6.2. They show:

– the throughput of *athletics* is constrained by the database service rate of a single node. Athletics and cycling activities demonstrate loss of throughput, but less than shown for the distributed database component alone. This may indicate the effect of the middleware queue.
– the throughput of *cycling* is constrained by the ratio between the input rates of athletics and cycling, as it was with the shared queue component. The

$$a = 1.0$$
$$c = 1.0$$
$$q = 100.0$$
$$db = 5.0$$

$$Website \stackrel{def}{=} (athletics, a).Website + (cycling, c).Website$$
$$Q_0 \stackrel{def}{=} (athletics, \top).Q_A + (cycling, \top).Q_C$$
$$Q_A \stackrel{def}{=} (queueA, \top).Q_0$$
$$Q_C \stackrel{def}{=} (queueC, \top).Q_0$$

$$DB_1 \stackrel{def}{=} (queueA, q).DBsrv_1$$
$$DBsrv_1 \stackrel{def}{=} (dbsrv1, db).DB_1$$
$$DB_2 \stackrel{def}{=} (queueC, q).DBsrv_2$$
$$DBsrv_2 \stackrel{def}{=} (dbsrv2, db).DB_2$$
$$Service_1 \stackrel{def}{=} (dbsrv1, db).Service_1$$
$$Service_2 \stackrel{def}{=} (dbsrv2, db).Service_2$$

$$Website \underset{\substack{athletics \\ cycling}}{\bowtie} Q_0[10] \underset{\substack{queueA \\ queueC}}{\bowtie} DB_1 \parallel DB_2 \underset{\substack{dbsrv1 \\ dbsrv2}}{\bowtie} Service_1 \parallel Service_2$$

**Fig. 17.** Shared queue and distributed database

behaviour of the queue appears to be the most significant when combined into a system.
– the database node throughput follows the throughput of each sport activity, i.e. the partitioning strategy routes all the demand onto the node handling that sport.

**Table 5.** Shared queue and distributed database experimental results

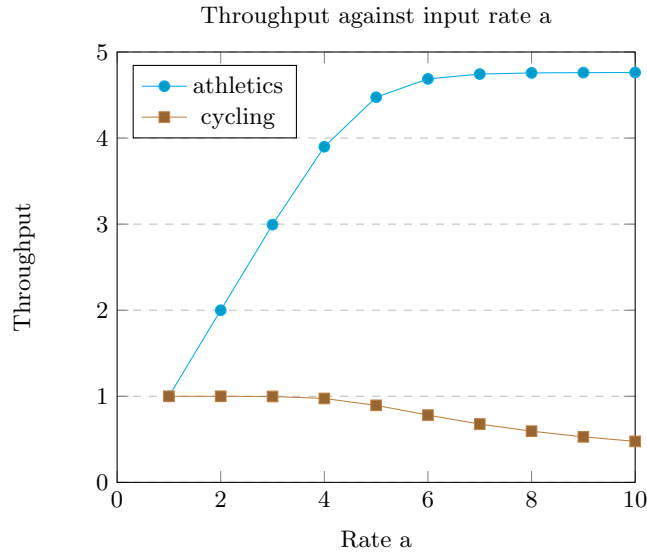| Rate | | | | Throughput | | | |
|------|-----------|---------|-------|--------|--------|--------|--------|
| a | athletics | cycling | ratio | queueA | queueC | dbsrv1 | dbsrv2 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 1 | 2 | 2 | 1 | 2 | 1 |
| 3 | 2.99 | 1 | 3 | 2.99 | 1 | 2.99 | 1 |
| 4 | 3.9 | 0.97 | 4 | 3.9 | 0.97 | 3.9 | 0.97 |
| 5 | 4.47 | 0.89 | 5 | 4.47 | 0.89 | 4.47 | 0.89 |
| 6 | 4.69 | 0.78 | 6 | 4.69 | 0.78 | 4.69 | 0.78 |
| 7 | 4.74 | 0.68 | 7 | 4.74 | 0.68 | 4.74 | 0.68 |
| 8 | 4.76 | 0.59 | 8 | 4.76 | 0.59 | 4.76 | 0.59 |
| 9 | 4.76 | 0.53 | 9 | 4.76 | 0.53 | 4.76 | 0.53 |
| 10 | 4.76 | 0.48 | 10 | 4.76 | 0.48 | 4.76 | 0.48 |

Throughput against input rate a



**Fig. 18.** Shared queue and distributed database - sport throughput

## 6.3   Shared queue and distributed database with replication

The final system model (Figure 6.3) combines a shared queue with a distributed database using replication, with one replica of each data partition on the 'next' node using consistent hashing. As before the website sends ticket requests via a shared queue to a worker application, which dequeues them for the database.

The PEPA model (Figure 6.3) combines the shared queue and distributed database with replication components, so that there is now an additional queue state $Q_D$ for holding *diving* ticket requests, and there are three data node processes. $DB_1$ is able to service *athletics* and *cycling* requests, $DB_2$ handles *cycling* and *diving*, and $DB_3$ handles *athletics* and *diving*. As with the component node, the model's assumption is that each data node handles each supported sport with equal probability. Again the model uses a queue length of N=10 and maximum queue worker rate of $q = 100.0$.
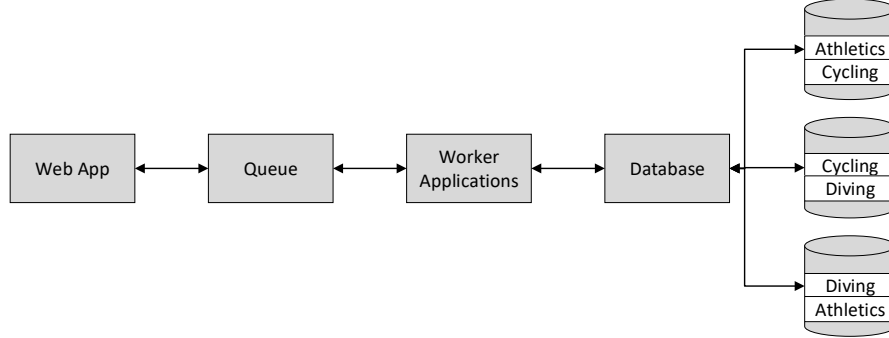
**Fig. 19.** Distributed database with replication architecture

$$a = 1.0$$
$$c = 1.0$$
$$d = 1.0$$
$$q = 100.0$$
$$db = 5.0$$

$$Website \stackrel{def}{=} (athletics, a).Website + (cycling, c).Website + (diving, d).Website$$
$$Q_0 \stackrel{def}{=} (athletics, \top).Q_A + (cycling, \top).Q_C + (diving, \top).Q_D$$
$$Q_A \stackrel{def}{=} (queueA, \top).Q_0$$
$$Q_C \stackrel{def}{=} (queueC, \top).Q_0$$
$$Q_D \stackrel{def}{=} (queueD, \top).Q_0$$

$$DB_1 \stackrel{def}{=} (queueA, q).DBsrv_1 + (queueC, q).DBsrv_1$$
$$DBsrv_1 \stackrel{def}{=} (dbsrv1, \top).DB_1$$
$$DB_2 \stackrel{def}{=} (queueC, q).DBsrv_2 + (queueD, q).DBsrv_2$$
$$DBsrv_2 \stackrel{def}{=} (dbsrv2, \top).DB_2$$
$$DB_3 \stackrel{def}{=} (queueD, q).DBsrv_3 + (queueA, q).DBsrv_3$$
$$DBsrv_3 \stackrel{def}{=} (dbsrv3, \top).DB_3$$
$$Service_1 \stackrel{def}{=} (dbsrv1, db).Service_1$$
$$Service_2 \stackrel{def}{=} (dbsrv2, db).Service_2$$
$$Service_3 \stackrel{def}{=} (dbsrv3, db).Service_3$$

$$Website \underset{\substack{cycling \\ diving}}{\overset{\bowtie}{athletics}} Q_0 [10] \underset{\substack{queueC \\ queueD}}{\overset{\bowtie}{queueA}} DB_1 \parallel DB_2 \parallel DB_3 \underset{\substack{dbsrv2 \\ dbsrv3}}{\overset{\bowtie}{dbsrv1}} Service_1 \parallel Service_2 \parallel Service_3$$

**Fig. 20.** Shared queue and distributed database with replication

Experiments are performed in the Eclipse plugin with the usual input rates. The resulting steady state throughputs are shown in Table 6, in Figure 6.3 as a chart comparing *athletics* and *cycling* throughput (as the cycling and diving throughputs are identical), and in Figure 6.3 showing the throughput of the database nodes. They show:

– the throughput of *athletics* is still constrained but is greater than that of a single database node. The demand is shared between both data nodes supporting athletics requests.
– the throughput of *cycling* (and diving) is constrained by the ratio between the input rates of athletics and cycling. In the component model, *cycling* and *diving* were impacted by athletics as they both shared a node with athletics tickets. In the system model, the queue effect appears to outweigh this.
– the database node throughput of the two nodes supporting athletics requests is equal, and both are higher than the remaining data node that supports only cycling and diving. The throughput of this node increases with athletics throughput up to a point, suggesting the node is handling an increasing proportion of the demand for cycling and diving tickets (but that this demand becomes constrained by the queue effect).

**Table 6.** Shared queue and distributed database with replication experimental results

| Rate | | | | | Throughput | | | | | |
|------|-----------|--------|--------|-------|--------|--------|--------|-------|-------|-------|
| a | athletics | cycling | diving | ratio | queueA | queueC | queueD | dbsrv1 | dbsrv2 | dbsrv3 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 1 | 1 | 2 | 2 | 1 | 1 | 1.46 | 1.08 | 1.46 |
| 3 | 3 | 1 | 1 | 3 | 3 | 1 | 1 | 1.92 | 1.16 | 1.92 |
| 4 | 4 | 1 | 1 | 4 | 4 | 1 | 1 | 2.38 | 1.24 | 2.38 |
| 5 | 5 | 1 | 1 | 5 | 5 | 1 | 1 | 2.84 | 1.32 | 2.84 |
| 6 | 5.98 | 1 | 1 | 6 | 5.98 | 1 | 1 | 3.29 | 1.4 | 3.29 |
| 7 | 6.93 | 0.99 | 0.99 | 7 | 6.93 | 0.99 | 0.99 | 3.72 | 1.47 | 3.72 |
| 8 | 7.76 | 0.97 | 0.97 | 8 | 7.76 | 0.97 | 0.97 | 4.09 | 1.51 | 4.09 |
| 9 | 8.4 | 0.93 | 0.93 | 9 | 8.4 | 0.93 | 0.93 | 4.38 | 1.51 | 4.38 |
| 10 | 8.83 | 0.88 | 0.88 | 10 | 8.83 | 0.88 | 0.88 | 4.56 | 1.47 | 4.56 |
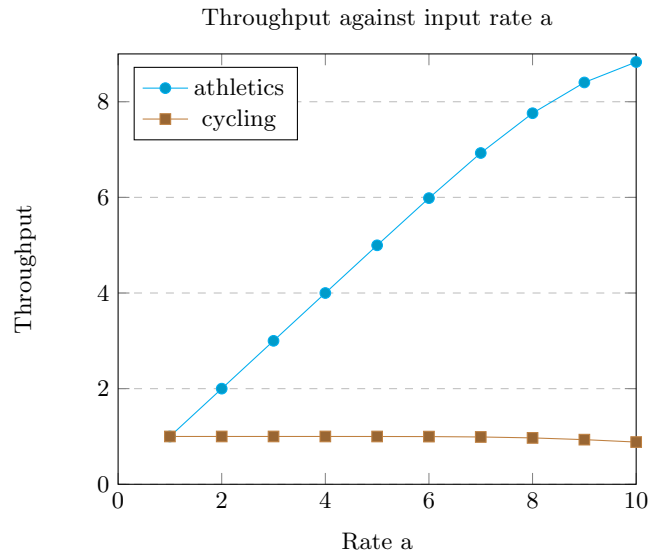
Throughput against input rate a



**Fig. 21.** Shared queue and distributed database with replication - sport throughput
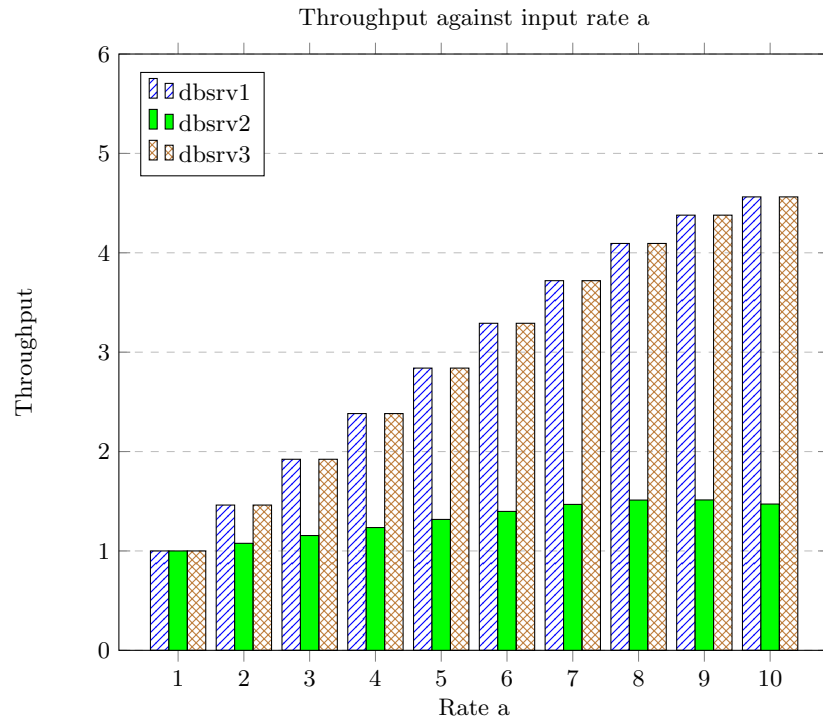
Throughput against input rate a



**Fig. 22.** Shared queue and distributed database with replication - database throughput

### 6.4 Comparison

As the experiments on the three system models all used the same range of input rates *a* and *c*, and the throughput of *athletics* and *cycling* requests was recorded for each system, then their experimental results may be compared with each other. (Caveat: only the distributed database with replication model has the additional *diving* activity, but its throughput was identical to diving). The *db* rate of the simple microservices model as been set back to 5.0, the same as the other models, and the numerical results of each system model test are compared in Table 7. If the models prove to be good representations of built systems, then such comparison shows the strengths and weaknesses of particular distributed system architectures and provides information for decision-making in their selection during design.

**Table 7.** Comparison of system results

| Rate | Simple Microservices | | Queue + Distributed DB | | Queue + DB with Replication | |
|------|----------|---------|----------|---------|----------|---------|
| a    | athletics | cycling | athletics | cycling | athletics | cycling |
| 1    | 0.96 | 0.96 | 1    | 1    | 1    | 1    |
| 2    | 1.76 | 0.96 | 2    | 1    | 2    | 1    |
| 3    | 2.39 | 0.96 | 2.99 | 1    | 3    | 1    |
| 4    | 2.87 | 0.96 | 3.9  | 0.97 | 4    | 1    |
| 5    | 3.23 | 0.96 | 4.47 | 0.89 | 5    | 1    |
| 6    | 3.5  | 0.96 | 4.69 | 0.78 | 5.98 | 1    |
| 7    | 3.71 | 0.96 | 4.74 | 0.68 | 6.93 | 0.99 |
| 8    | 3.87 | 0.96 | 4.76 | 0.59 | 7.76 | 0.97 |
| 9    | 4.01 | 0.96 | 4.76 | 0.53 | 8.4  | 0.93 |
| 10   | 4.11 | 0.96 | 4.76 | 0.48 | 8.83 | 0.88 |

*Athletics Throughput.* The comparison shows that the actual throughput of the skewed demand for athletics tickets is lowest for the simple microservices system. Introducing queue middleware to the system gives a throughput much closer to the maximum service rate of a database node. Using a distributed database with replication has a load sharing effect so that this system comes closest of all to a throughput that meets the demand.

*Cycling Throughput.* In contrast, the cycling throughput is overall higher for the simple microservices system. At lower levels of athletics demand, the queue-based systems win out, but as the skewed demand increases then it begins to have an impact on the cycling throughput in these systems. As the microservices architecture isolates cycling from athletics, it remains at a constant rate.

# 7  Built systems

## 7.1  Approach

To test the system models, real systems using those architectures are built and their throughputs measured under simulated workloads.

**Designs.** All three systems were developed in Java and deployed on Microsoft Azure virtual machines. The system designs discuss their specifics, but the common feature is that all three systems use a Cassandra database or databases with the same ticket schema design, given in Figure 7.1, with 500 tickets of each sport.

Cassandra was chosen as the distributed database models were designed for consistent hashing (most relevant for the model using replication). Although the simple microservices model need not have used Cassandra databases, reusing the schema design and database servers was an economical choice, and it illustrates how different architectures may arrange many of the same components.

The source code for the systems and related tools is hosted on GitHub [27].

```
/*
 * Ticket table schema
 *
 * int id - unique ticket id
 * varchar sport - type of sport
 * int day - day of event
 * int seat - seat number
 * varchar owner - name of booked ticket owner
 *
 * The partition key is sport
 * The clustering columns are owner, day, id
 */

CREATE TABLE IF NOT EXISTS ticket (
        id int,
        sport varchar,
        day int,
        seat int,
        owner varchar,
        PRIMARY KEY (sport, owner, day, id)
) WITH comment = 'Tickets';
```

**Fig. 23.** Cassandra database ticket schema

**Workload Simulation.** The workload from a web application and its users was simulated using Apache JMeter [4], a Java application for load testing. JMeter was originally designed for testing web applications and all the systems have RESTful APIs over HTTP (for the simple microservices architecture, the Java Spring APIs; for the shared queue architectures, the Microsoft Azure Storage Queue REST APIs). JMeter test plans are composed of thread groups, where the number of threads and test executions are specified, samplers such as HTTP Request, and timers to determine the delay between each test execution. Increasing the number of threads used in the test increases the required demand. JMeter's Poisson Random Timer is used to simulate the negative exponential distribution [12] that PEPA steady state solutions assume.

**Measurement.** Measurement of throughputs in the built systems is required to produce results that can be compared with the PEPA model experimental results.

*Measurands.* The measurands are the throughputs of athletics and cycling requests in the worker applications, and control requests that do not access the database in order to show any system overheads. Where applicable, measurands include the throughputs of diving requests and database queries.

*Measurement method.* The worker applications are measured using Coda Hale Metrics [8], a Java library providing instrumentation. The instrument used is a *Meter*, that measures the rate at which events occur as mean and moving average rates. Database throughputs are measured by enabling the built-in Cassandra metric ThreadPools.CompletedTasks.request.ReadStage (the total count of completed read queries). In both cases the metrics are logged every 10 seconds.

*Measurement procedure.* The largest 1-minute moving average over a test run is extracted by a Python script. For the worker applications, the 1-minute average is provided by the Meter instrument. For Cassandra metric files, the average must be calculated, again using a Python script. Each experiment is carried out five times and the mean is taken of the five sets of results.

### 7.2   Simple microservices

**Design.** The simple microservices architecture from Figure 6.1 was deployed on four Azure Ubuntu virtual machines as shown in Table 8. There are two completely separate Cassandra databases, one containing Athletics tickets, one for Cycling tickets. There are two instances of the *SimplemicroApplication* worker application, each running on a separate virtual machine and each connecting to one of the databases.

SimplemicroApplication was built using Java Spring [23], a collection of frameworks including model-view-controller (used to implement RESTful APIs for the application) and data access (which supports connection to many databases including Cassandra). The application has two RESTful APIs:

\search, which takes a sport parameter (Athletics or Cycling), queries the database for all matching tickets, and records metrics.

\control, which doesn't access the database, but still records metrics.

**Table 8.** Simple microservices Azure VMs

| VM | Specification | Application |
|---|---|---|
| worker1 | DS1 v2 (1 core, 3.5 GB) | SimplemicroApplication |
| worker2 | DS1 v2 (1 core, 3.5 GB) | SimplemicroApplication |
| db1 | F1s (1 core, 2 GB) | Cassandra |
| db2 | F1s (1 core, 2 GB) | Cassandra |

**Workload Simulation.** The application was tested using JMeter with two thread groups running in parallel, Cycling with a constant 10 threads (users) and Athletics ramping up from 10-100 in steps of 10. Each thread group had a Poisson random timer with a lambda value of 500 milliseconds and a loop count of 500 requests, ensuring several minutes worth of samples and therefore a number of rolling 1-minute averages. The test plan was used against the \control APIs for tuning and calibration, showing that the constant demand was approximately 19 requests per second and the skewed demand ranged from 19-190 requests per second (again, approximately, given that a the distribution is random). The test plan was then applied to the \search API with the same parameters.

**Results.** The results are given numerically in Table 9 and shown graphically in Figure 7.2 (where the 'control' plot is the control throughput from the athletics virtual machine). The athletics worker throughput using \search was measured to be limited to 130 queries per second, compared to the control throughput on the same virtual machine i.e. that is the limit imposed by the database. Using Cassandra's packaged cassandra-stress load testing tool on the database server shows that it is capable of much higher query rate, which suggests that using Java Spring Data adds overheads to the database requests (perhaps, given the use of sessionless RESTful APIs, the main overhead is setting up a connection session to Cassandra each time). The PEPA database model has already been acknowledged to be an abstraction which does not include such factors, and the scaled rate was fed back into the model's database service rate.

The cycling throughput is more isolated from athletics than the queue-based systems, but there is a slow downward trend as athletics demand increases. The only point at which the two microservices connect is at JMeter, so this may be the effect of increased demand on JMeter itself, although the control throughput results from the cycling virtual machine do not clearly demonstrate the same trend. Another potential cause is co-residency of the virtual machines.

**Table 9.** Simple microservices experimental results

| | Athletics worker | | | Cycling worker | |
|-------|---------|--------|-------|---------|--------|
| users | control | search | users | control | search |
| 10 | 19.29 | 17.92 | 10 | 19.25 | 18.08 |
| 20 | 37.67 | 34.39 | 10 | 18.99 | 17.61 |
| 30 | 56.44 | 51.11 | 10 | 18.96 | 17.52 |
| 40 | 74.61 | 67.13 | 10 | 18.75 | 17.4 |
| 50 | 92.59 | 81.54 | 10 | 18.73 | 17.18 |
| 60 | 111.6 | 95.52 | 10 | 18.7 | 16.82 |
| 70 | 131.43 | 111.3 | 10 | 18.83 | 16.93 |
| 80 | 150.49 | 120.7 | 10 | 18.85 | 16.33 |
| 90 | 168.29 | 130.09 | 10 | 18.83 | 16.06 |
| 100 | 185.85 | 134.01 | 10 | 18.72 | 15.94 |

Throughput against athletics demand
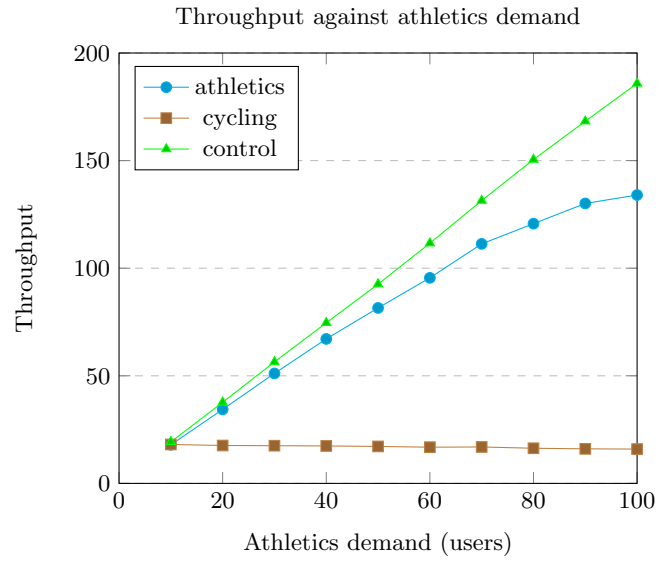
**Fig. 24.** Simple microservices experimental results

### 7.3   Shared queue and distributed database

**Design.** The shared queue and distributed database architecture from Figure 6.2 was deployed on three Azure Ubuntu virtual machines as shown in Table 10, and using a single Azure Storage Queue instance. Azure queues provide their own RESTful API access and JMeter was used to populate the queue with Athletics, Cycling or Control ticket requests.

The *QueueWorker* application was built as a multithreaded application running on a single, multi-core virtual machine. It dequeues every request (Athletics, Cycling or Control tickets) from the shared Azure Storage Queue. This asynchronous middleware would be best suited to the 'return ticket' operation from the use case, but to ensure that a usable Cassandra metric was available for measuring database throughput, an Athletics or Cycling request was interpreted as a 'search ticket' operation. A database select of all tickets for the matching sport is carried out first and the metric is recorded if the query returns results. As before the Control request does not access the database but records metrics. When connecting to the Cassandra database, the QueueWorker application uses a round-robin algorithm to select the coordinator node, to ensure that any additional overheads incurred are shared equally between both nodes.

As populating the queue with JMeter and processing it with the queue worker application are decoupled, it was possible to run QueueWorker on a prepopulated queue to determine its maximum throughput i.e. regardless of the incoming demand. Using this technique suggested that maximum performance came with QueueWorker running with 16 threads.

The Cassandra database was distributed using a keyspace ('Distributed') with SimpleStrategy, replication_factor=1 onto two nodes each on one virtual machine. Cassandra's partitioning places Athletics tickets on one node, and Cycling tickets on the other. This was validated using nodetool getendpoints, which shows which node hosts records from a given keyspace matching a given partition key e.g. nodetool getendpoints Distributed ticket Athletics.

Note that without the overheads of starting a new Cassandra database session for each request, it was necessary to slow Cassandra down by turning on tracing for 100% of queries using nodetool settraceprobability 1.0. Load testing the database alone with cassandra-stress using 16 threads and a query for all athletics tickets suggested a maximum database throughput of 475 operations per second.

**Table 10.** Shared queue and distributed DB Azure VMs

| VM | Specification | Application |
|----|---------------|-------------|
| qworkers | DS3 v2 Promo (4 cores, 14 GB) | QueueWorker (16 threads) |
| db1 | F1s (1 core, 2 GB) | Cassandra |
| db2 | F1s (1 core, 2 GB) | Cassandra |

**Workload Simulation.** The application was tested using JMeter with two thread groups running in parallel, Cycling with a constant 15 threads and Athletics ramping up from 15-150 in steps of 15. Each thread group had a Poisson random timer with a lambda value of 100 milliseconds and a loop count of 1500 requests, ensuring enough samples for a number of rolling 1-minute averages. Again there was a control version of test plan with the same variables, but sending Control ticket requests to the queue rather than Athletics and Cycling tickets. This produces constant demand of approximately 95 requests per second and skewed demand from 95-950 requests per second. Scaling these results with the cassandra-stress measurement about to the model input range of 1-10 suggested keeping the *db* rate at 5.0.

**Results.** Table 11 shows the numerical results and Figure 7.3 shows the plots of athletics, cycling and control throughputs for the same range of athletics demands stated as the number of JMeter threads or users.

The total throughput of *athletics* and *cycling* quickly falls behind the control throughput, and examination of the athletics throughput in particular shows that it is constrained by the database performance of a single node. Note that the constrained throughout is significantly less than the maximum rate shown by cassandra-stress. Interestingly the maximum total throughput is still less than that value but much closer to it, so the limit shown by cassandra-stress may be based on the whole cluster even though the tickets only appear on one node.

The *cycling* throughput is constrained in proportion to the athletics *demand*, not the actual athletics throughput, so that as the athletics throughput is limited but the demand continues to decrease, then the cycling throughput decreases in the ratio of their respective demands (though the ratio is not exact). The behaviour of the built system matches the prediction of the model. The model had a very small queue size, and an actual Azure Storage Queue is practically unlimited, so this result was not certain.

Finally the database node throughput follows the throughput of each sport activity. The partitioning strategy placed the athletics tickets onto *db2*, as discovered using nodetool getendpoints, and the throughput on this node increases with the skewed athletics demand.

**Table 11.** Shared queue and distributed DB experimental results

| Rate | Throughput | | | | | | |
|------|-----------|--------|--------|---------|-------|--------|--------|
| (users) | athletics | cycling | total | control | ratio | db1 | db2 |
| 15 | 94.55 | 93.83 | 188.39 | 187.03 | 1.01 | 97.88 | 98.96 |
| 30 | 186.02 | 97.53 | 283.55 | 272.69 | 1.91 | 101.46 | 194.23 |
| 45 | 282.04 | 95.56 | 377.6 | 372.48 | 2.95 | 97.66 | 293.27 |
| 60 | 346.47 | 95.08 | 441.55 | 474.85 | 3.64 | 96.13 | 362.94 |
| 75 | 358 | 86.33 | 444.33 | 561.37 | 4.15 | 84.96 | 365.02 |
| 90 | 366.9 | 71.46 | 438.36 | 637.29 | 5.13 | 72.04 | 373.2 |
| 105 | 369.29 | 60.7 | 429.99 | 746.53 | 6.08 | 61.39 | 376.63 |
| 120 | 378.14 | 53.47 | 431.61 | 793.76 | 7.07 | 53.93 | 385.62 |
| 135 | 374.62 | 45.31 | 419.93 | 856.46 | 8.27 | 45.51 | 381.08 |
| 150 | 378.9 | 41.38 | 420.28 | 953.17 | 9.16 | 41.9 | 388.41 |
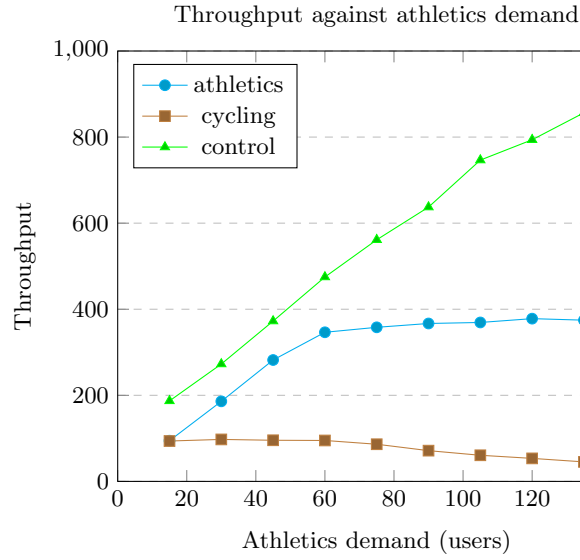


**Fig. 25.** Shared queue and distributed DB experimental results

### 7.4   Shared queue and distributed database with replication

**Design.** The shared queue and distributed database with replication architecture from Figure 6.3 was deployed on four Azure Ubuntu virtual machines as shown in Table 12, again using a single Azure Storage Queue instance.

The *QueueWorker* application was reused (as well as handling Athletics, Cycling or Control ticket requests, the application handles Diving requests) and run with 16 threads as before.

The Cassandra database was distributed using a keyspace ('Replicated') with SimpleStrategy, replication_factor=2 onto three nodes each on one virtual machine. Cassandra's partitioning places Athletics, Cycling and Diving tickets on different nodes with each node also containing replicas of one other ticket type (it was necessary to use ByteOrderedPartitioner to force this). This is verified using nodetool getendpoints e.g. nodetool getendpoints Distributed ticket Athletics.

| Node | Primary | Replica |
|------|---------|---------|
| db1 | Athletics | Cycling |
| db2 | Cycling | Diving |
| db3 | Diving | Athletics |

Again, it was necessary to slow Cassandra down by turning on tracing for 100% of queries.

Load testing the database with cassandra-stress using 16 threads suggested a maximum database throughput of 600 operations per second. This suggests that replicating tickets on two database nodes increases the possible throughput, but it hasn't doubled the previous performance of 475 ops per second. The model keeps the *db* rate of 5.0 based on that 475 ops per second single-node measurement.

**Table 12.** Shared queue and distributed DB with replication Azure VMs

| VM | Specification | Application |
|------|---------------|-------------|
| qworkers | DS3 v2 Promo (4 cores, 14 GB) | QueueWorker (16 threads) |
| db1 | F1s (1 core, 2 GB) | Cassandra |
| db2 | F1s (1 core, 2 GB) | Cassandra |
| db3 | F1s (1 core, 2 GB) | Cassandra |

**Workload Simulation.** The application was tested using JMeter with three thread groups running in parallel, Cycling and Diving with a constant 15 threads and Athletics ramping up from 15-150 in steps of 15. The parameters were set to the same values as the previous JMeter test plan.

**Results.** Table 13 shows the numerical results. Figure 7.4 shows the plots of athletics, cycling and control throughputs for the same range of athletics demands. As for the model, diving throughput is not shown on the chart as it closely matches cycling throughput. Figure 7.4 shows the database throughputs for all three nodes.

The control throughput begins to taper off as it gets closer to 1000 requests per second. Azure queues have a limit of up to 2000 messages per second, so simultaneously queueing and dequeuing requests at the same rate suggests a limit of 1000 messages per second. The total throughput of *athletics*, *cycling* and *diving* still falls behind control throughput, and again the maximum total is very close to the cassandra-stress limit of 600 requests per second. Both the total and athletics throughputs are significantly higher than for the system without replication, as predicted by the model.

Again the model correctly predicted that the *cycling* and *diving* throughputs would be constrained in a ratio to the athletics throughput that matches (or is close to) the ratio between their demands.

However, the database node throughputs are very different to those from the model. While the *db1* and *db3* nodes that host athletics tickets do show higher throughputs than the node which does not, the throughputs were not shared equally between those two nodes. Neither was the allocation random. The average of all five sets of tests showed *db3* taking many more requests. This, and the cassandra-stress result showing that the athletics query performance did not double when adding a replica to another node, indicates that Cassandra's performance is more complex than the abstract database model.

**Table 13.** Shared queue and distributed database with replication experimental results

| Rate | Throughput | | | | | | | | |
|------|-----------|---------|--------|-------|---------|-------|-------|--------|--------|
| (users) | athletics | cycling | diving | total | control | ratio | db1 | db2 | db3 |
| 15 | 111.97 | 112.08 | 111.92 | 335.97 | 335.2 | 1 | 137.26 | 167.58 | 151 |
| 30 | 219.25 | 113.96 | 113.93 | 447.14 | 439.82 | 1.92 | 187.74 | 149.96 | 208.57 |
| 45 | 336.25 | 114.5 | 114.5 | 565.26 | 561.71 | 2.94 | 194.73 | 184.98 | 301.59 |
| 60 | 379.1 | 114.14 | 114.18 | 607.41 | 643.45 | 3.32 | 192.23 | 171.2 | 307.34 |
| 75 | 393.12 | 95.15 | 95.24 | 583.51 | 772.81 | 4.13 | 186.57 | 161.65 | 309.76 |
| 90 | 404.55 | 78.76 | 78.56 | 561.87 | 847.28 | 5.14 | 175.88 | 142.35 | 327.07 |
| 105 | 415.51 | 67.68 | 67.69 | 550.88 | 891.84 | 6.14 | 167.31 | 126.33 | 339.73 |
| 120 | 416.2 | 59.71 | 59.52 | 535.42 | 920.51 | 6.97 | 172.79 | 112.52 | 331.42 |
| 135 | 414.05 | 51.88 | 52.06 | 517.98 | 948.76 | 7.98 | 171.73 | 97.07 | 332.88 |
| 150 | 422.42 | 46.39 | 46.13 | 514.94 | 961.01 | 9.11 | 170.61 | 88.64 | 341.23 |
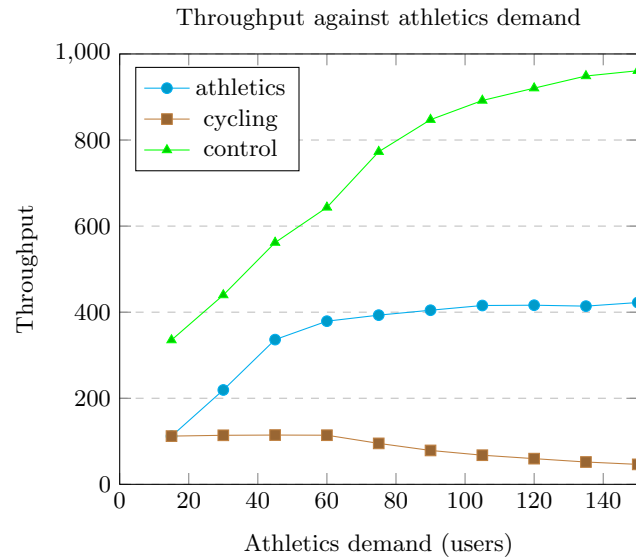
Throughput against athletics demand



**Fig. 26.** Shared queue and distributed database with replication - sport throughput

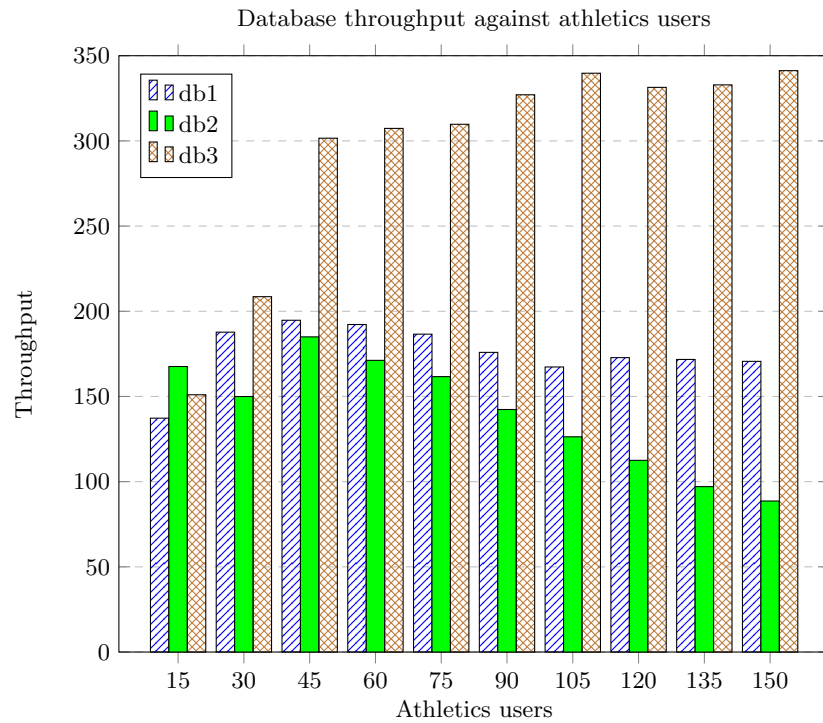Database throughput against athletics users



**Fig. 27.** Shared queue and distributed database with replication - database throughput

## 8  Conclusions and Future Work

PEPA can be used to build models of complex systems using distributed, cloud-based technologies. Experiments can be run in the PEPA Eclipse plugin to produce steady state throughput results, and make predictions of how the real, built systems will behave under conditions of skewed demand.

The models are useful for qualitative prediction. They successfully predicted:

- that the microservices architecture would better isolate skewed demand from the other components.
- (less trivially) when a shared queue is used in combination with a distributed database, once the throughput of resources with skewed demand is choked, the throughput of the other resources was choked in proportion to the relative demand between them and the skewed resource. This even though the model's queue size is much smaller than an actual Cloud service queue.
- where throughput would be routed through a distributed database when using a particular partitioning strategy.
- when using a distributed database with replication, there would also be throughput at the replica node, and that therefore the overall throughput of the skewed demand resource would be higher than for a distributed database without replication.

However, the models were less successful at quantitative predictions. Simple microservices was still best at isolating demand but built system results showed that there was some impact on the other resources. When using replication, the throughput was not spread evenly (or randomly) between database nodes, and this also meant that the system was unable to satisfy as much demand as predicted. The models were too abstract to describe and predict the behaviour inside the database engine, or the overheads incurred by worker application design choices. This meant that the maximum apparent database service rates for actual systems were very different for microservices and the shared queue distributed database models, despite both using the same queries against Cassandra on the same virtual machines. This means that while the models suggested that they could be used to compare different architectures directly, and to make predictions about which would make the best use of the resources available, the real systems did not bear this out. The models as they stand are not therefore suitable for right-sizing applications.

It is argued that the models have demonstrated potential for further work in this application area.

### 8.1  Improved Models

The real Cassandra database behaviour was shown to be more complex than described by these simple models. One area of future work might be to build on the abstract database model to exhibit the true behaviour more closely. Care must be taken however not to customise the model so much that it is no longer useful for other database implementations.

## 8.2   System Experiments

The existing models may be used for further experimentation to determine how the currently modelled technologies may be reconfigured to produce better system architectures. Partitioning strategies normally aim to use all a distributed database's nodes equally. In such a strategy, what is the impact of skewed demand? Also, the shared queue led to skewed demand having a significant impact on the other resources, and a valuable subject of further study would be how other queueing strategies might isolate that demand while retaining the advantages of asynchronous middleware. There are several queue models in [29] that would provide a good starting point, e.g. priority queues.

## 8.3   Unknown Skewed Demand

This paper did not consider systems where the source of skewed demand was unknown in advance. There is much interest in this problem, for example handling flash crowds in peer-to-peer file sharing networks [26]. An interesting area of future work might be in executing the models at runtime to use them for adaptive algorithms, e.g. to change the queueing policy on-demand. If the models were also improved to the point that they made better quantitative predictions, then they might also be used as an elastic scaling policy and compared with the performance of other right-sizing strategies; control theory, machine learning and other model based techniques including statistical.

## 8.4   New Models

It may also be possible to use PEPA to produce models for other distributed technologies. For example, a more 'natural' microservices architecture would partition the system by operation (Book, Search, Return) with a separate database for each, as shown in Figure 3.3. In this architecture the databases maintain eventual consistency via an event streaming application e.g. using Kafka. The model for this component may well be generalised to a model of publish/subscribe message oriented middleware.

# References

1. de Abranches, M.C., Solis, P.: An algorithm based on response time and traffic demands to scale containers on a cloud computing system. In: Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on. pp. 343–350. IEEE
2. Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. pp. 359–370. ACM (2004)
3. Apache: Apache cassandra (2017), http://cassandra.apache.org/, [Online; accessed 28-June-2017]
4. Apache: Apache jmeter (2017), http://jmeter.apache.org, [Online; accessed 28-June-2017]
5. Bryhni, H., Klovning, E., Kure, O.: A comparison of load balancing techniques for scalable web servers. IEEE network 14(4), 58–64 (2000)
6. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and experience 41(1), 23–50 (2011)
7. Cattell, R.: Scalable sql and nosql data stores. Acm Sigmod Record 39(4), 12–27 (2011)
8. Coda Hale: Metrics (2014), http://metrics.dropwizard.io/3.2.2/, [Online; accessed 28-June-2017]
9. Curry, E.: Message-oriented middleware. Middleware for communications pp. 1–28 (2004)
10. Dan Deeth, Sandvine: Hbo goes down (2014), http://www.internetphenomena.com/2014/03/hbo-goes-down/, [Online; accessed 15-March-2017]
11. Fang, W., Lu, Z., Wu, J., Cao, Z.: Rpps: a novel resource prediction and provisioning scheme in cloud data center. In: Services Computing (SCC), 2012 IEEE Ninth International Conference on. pp. 609–616. IEEE
12. Feitelson, D.G.: Workload modeling for computer systems performance evaluation (2015)
13. Gilly, K., Juiz, C., Puigjaner, R.: An up-to-date survey in web load balancing. World Wide Web 14(2), 105–131 (2011)
14. Götz, N., Herzog, U., Rettelbach, M.: Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. Performance evaluation of computer and communication systems pp. 121–146 (1993)
15. Hecht, R., Jablonski, S.: Nosql evaluation: A use case oriented survey. In: Cloud and Service Computing (CSC), 2011 International Conference on. pp. 336–341. IEEE
16. Hillston, J.: A compositional approach to performance modelling. Computers & Mathematics with Applications 32(6), 136 (1996)
17. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
18. Lewis, J., Fowler, M.: Microservices (2014), martinfowler.com, [Online; accessed 5-March-2017]

19. Lim, H.C., Babu, S., Chase, J.S.: Automated control for elastic storage. In: Proceedings of the 7th international conference on Autonomic computing. pp. 1–10. ACM
20. Mell, P., Grance, T.: The nist definition of cloud computing (2011)
21. Microsoft: Azure storage documentation (2017), https://docs.microsoft.com/en-us/azure/storage/, [Online; accessed 27-July-2017]
22. Nick Pearce, Telegraph: London olympics 2012: ticket site temporarily crashes as it struggles to cope with second-round demand (2011), http://www.telegraph.co.uk/sport/olympics/8595834/London-Olympics-2012-ticket-site-temporarily-crashes-as-it-struggles-to-cope-with-second-round-demand.html, [Online; accessed 2-March-2017]
23. Pivotal: Spring (2017), http://spring.io/, [Online; accessed 28-June-2017]
24. Posta, C.: Carving the java ee monolith into microservices: Prefer verticals not layers (2016), http://blog.christianposta.com/microservices/carving-the-java-ee-monolith-into-microservices-perfer-verticals-not-layers/, [Online; accessed 9-March-2017]
25. Posta, C.: The hardest part about microservices: Your data (2016), https://developers.redhat.com/blog/2016/08/02/the-hardest-part-about-microservices-your-data/, [Online; accessed 5-March-2017]
26. Rubenstein, D., Sahu, S.: Can unstructured p2p protocols survive flash crowds? IEEE/ACM Transactions on Networking 13(3), 501–512 (2005)
27. Shephard, S.: Performance modelling and simulation of skewed demand in complex systems (2017), https://github.com/sshephard2/skewed-modelling, [Online; accessed 28-June-2017]
28. The Next Web: itunes is down for many users around the world (2016), https://thenextweb.com/apple/2016/09/16/itunes-store-is-down-for-some-users, [Online; accessed 15-March-2017]
29. Thomas, N., Hillston, J.: Using Markovian process algebra to specify interactions in queueing systems. University of Edinburgh (1997)
30. Tribastone, M., Duguid, A., Gilmore, S.: The pepa eclipse plugin. ACM SIGMETRICS Performance Evaluation Review 36(4), 28–33 (2009)