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 [23]. HBO Go was brought down by demand for the finale of "True Detective" [10]. 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].

Some more examples, preferably from peer-reviewed papers? (p2p file sharing, multi-tenanting?)

It is claimed 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).

It is proposed that a selection of technologies may be modelled as simple components, that may be composed into more complex system models that make end to end predictions. When combining a middleware solution with a distributed

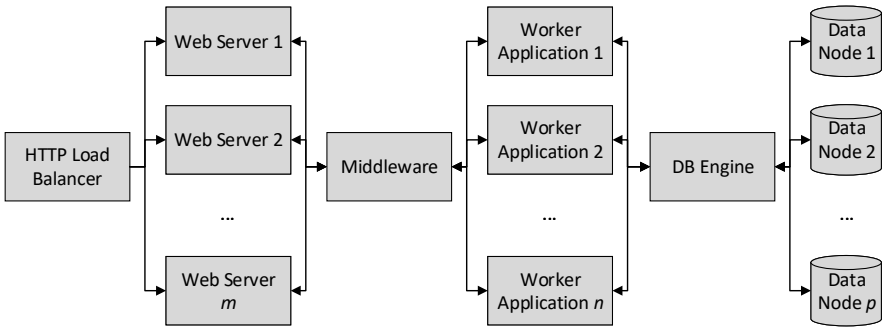
database, where is the system bottleneck? If there are levels of demand that cannot be met on a limited budget, and that therefore some components will no longer meet the required throughput, what is the impact on the remainder of the system?

The models will then be tested against actual systems, built using Java and the Java Spring framework [24], Cassandra [18][3] databases, and where appropriate Microsoft Azure Storage Queues [22]. These systems will be instrumented with Coda Hale Metrics [8] and measured under different scenarios of skewed demand, using Apache JMeter test plans [4].

2 Background

Consider a general OLTP application using a distributed architecture, as shown in Figure 1. 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 of the layers of this architecture.

Fig. 1. OLTP application distributed 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 [21]. 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.

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 using 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 skewed demand is unknown - where the areas of highest demand 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.

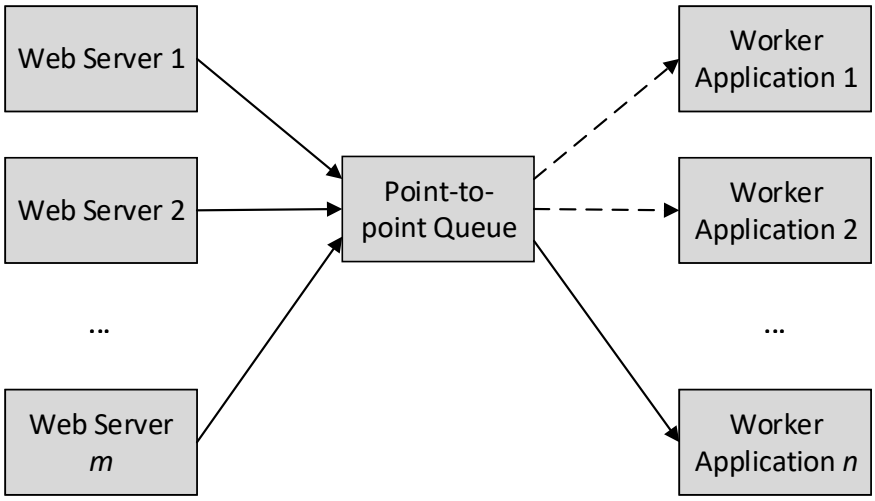
Out of Scope. The paper will not consider elastic scaling or HTTP load balancing. There is already a great deal of work in evaluating right-sizing strategies (minimising underutilisation and overutilisation of compute resources) for the former, e.g. [1], [11], [20]. HTTP load balancing is a relatively 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 [22], 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.

Fig. 2. Point-to-Point Queue Middleware

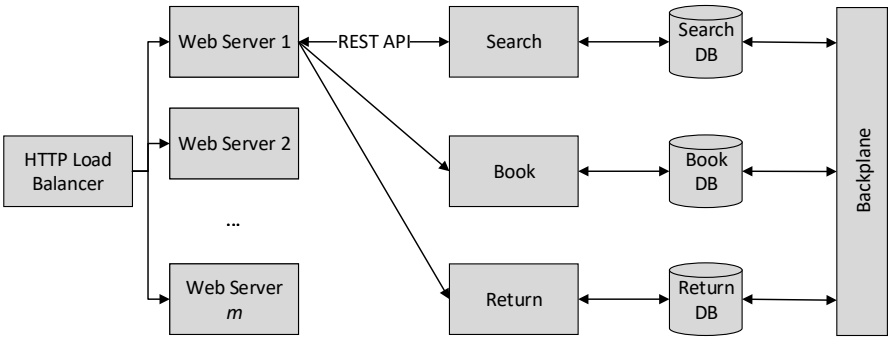


Many processes may send messages to a queue, and each message is received by one consumer - though it may be one of several consumers competing for messages from this queue. This competing consumer pattern (see Figure 2) offers a means of balancing load from the Web Servers between the Worker Applications in the ticketing use case.

3.3 Microservices

Microservice architecture is an approach to structuring applications as suites of small services, defined by business capability verticals rather than technological layers [19] [25]. Each of the use case requirements - search for tickets, book tickets, 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 [26] as shown in Figure 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 booking tickets was 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. 3. Microservices



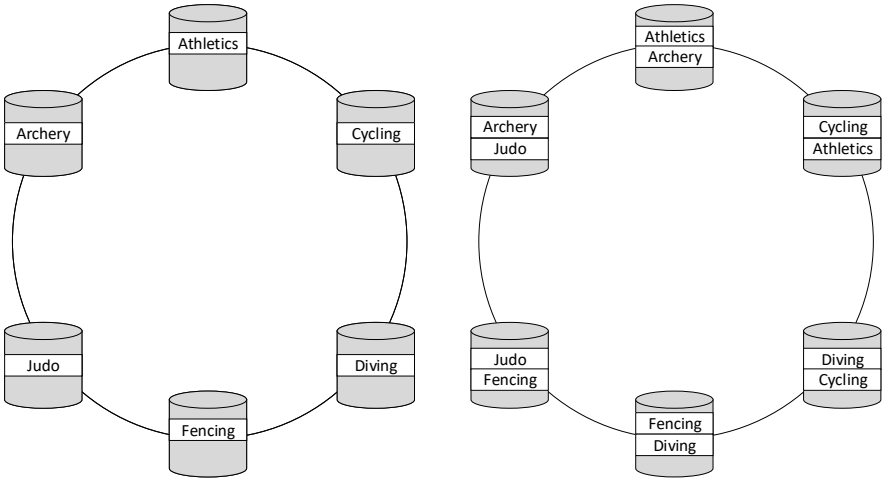
3.4 Distributed databases

Modern databases both SQL and NoSQL 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 a data node is overloaded by demand, 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. 4. 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 [18]. This is illustrated in Figure 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 language-based models (e.g. *CloudSim*) or mathematical language-based models (e.g. *Process Algebra*).

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 a full lower case name (or Greek letter) 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 α 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: $P \underset{L}{\bowtie} Q$ - 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 $P \underset{\emptyset}{\bowtie} Q$.

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):

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

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. Therefore there is no ordering guarantee e.g. the queue is not guaranteed to be First In First Out (FIFO). Actual cloud service queues do not necessarily implement FIFO, for example Azure Storage Queues [22] 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.

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.

Fig. 5. Shared queue PEPA model

$$
\begin{aligned}
a &= 1.0 - 10.0 \\
c &= 1.0 \\
s1 &= 5.0 \\
s2 &= 5.0
\end{aligned}
$$

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

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

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

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 6 shows the throughput of *athletics* for exponentially increasing queue lengths from 1 to 20. Figure 7 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 arrival and service processes are less coupled, and 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 model with sufficient decoupling.

Fig. 6. Shared queue experimental results - athletics
Throughput of athletics against input rate a for different queue lengths N

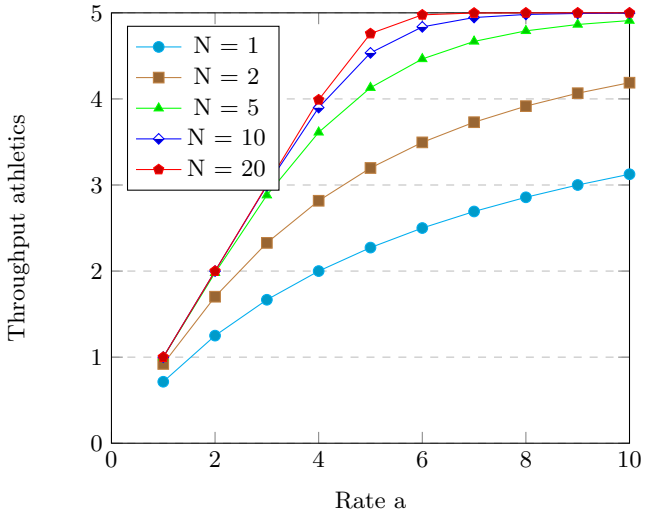


Fig. 7. Shared queue experimental results - cycling
Throughput of cycling against input rate a for different queue lengths N

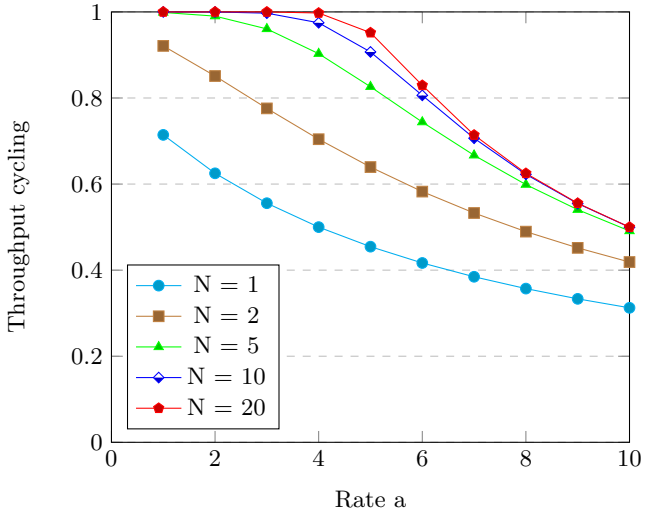


Table 1. Shared queue N=10 experimental results

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 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 8 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.

Fig. 8. Distributed database PEPA model

$$
\begin{aligned}
a &= 1.0 - 10.0 \\
c &= 1.0 \\
db &= 5.0
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Website} &\stackrel{def}{=} (\mathit{athletics}, a).\mathit{Website} + (\mathit{cycling}, c).\mathit{Website} \\
\mathit{DB}_1 &\stackrel{def}{=} (\mathit{athletics}, \top).\mathit{DBsrv}_1 \\
\mathit{DBsrv}_1 &\stackrel{def}{=} (\mathit{dbsrv1}, \top).\mathit{DB}_1 \\
\mathit{DB}_2 &\stackrel{def}{=} (\mathit{cycling}, \top).\mathit{DBsrv}_2 \\
\mathit{DBsrv}_2 &\stackrel{def}{=} (\mathit{dbsrv2}, \top).\mathit{DB}_2 \\
\mathit{Service}_1 &\stackrel{def}{=} (\mathit{dbsrv1}, db).\mathit{Service}_1 \\
\mathit{Service}_2 &\stackrel{def}{=} (\mathit{dbsrv2}, db).\mathit{Service}_2
\end{aligned}
$$

$$
\mathit{Website} \underset{\mathit{cycling}}{\overset{}{\bowtie}} \mathit{DB}_1 \parallel \mathit{DB}_2 \underset{\mathit{dbsrv2}}{\overset{}{\bowtie}} \mathit{Service}_1 \parallel \mathit{Service}_2
$$

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 9 shows the throughput of both *athletics* and *cycling* against the skewed input rate a, and Figure 10 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 of the demand onto the node handling that sport.

Fig. 9. Distributed database without replication - sport throughput

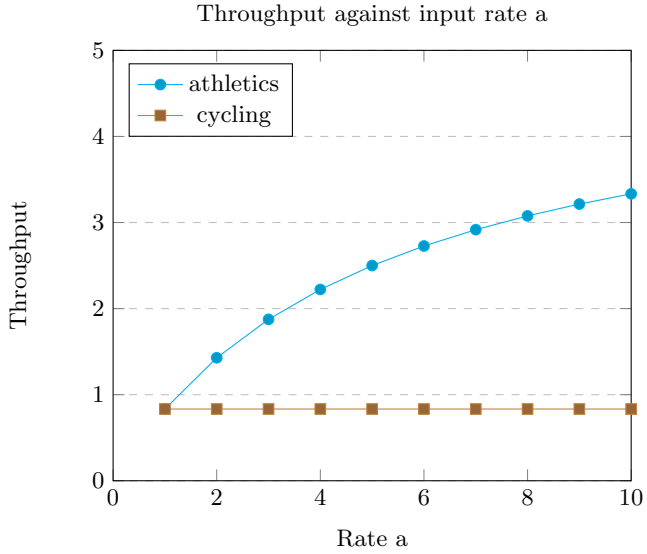


Fig. 10. Distributed database without replication - database throughput

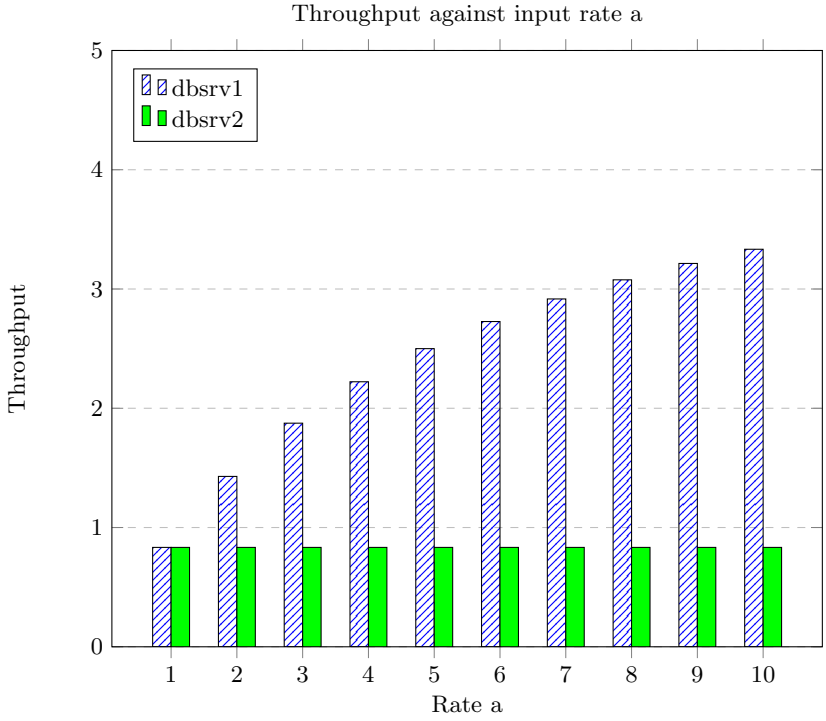


Table 2. Distributed database without replication experimental results

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 11. 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 12 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 13 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.

Fig. 11. Distributed database with replication PEPA model

$$
\begin{aligned}
a &= 1.0 - 10.0 \\
c &= 1.0 \\
d &= 1.0 \\
db &= 5.0
\end{aligned}
$$

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

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

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

Fig. 12. Distributed database with replication - sport throughput

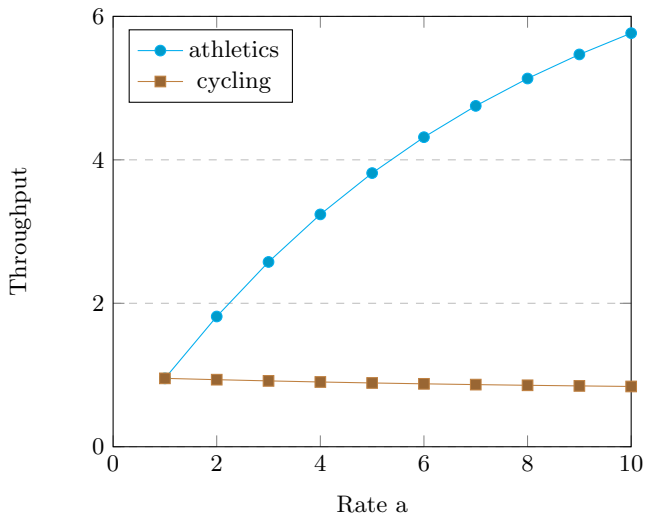
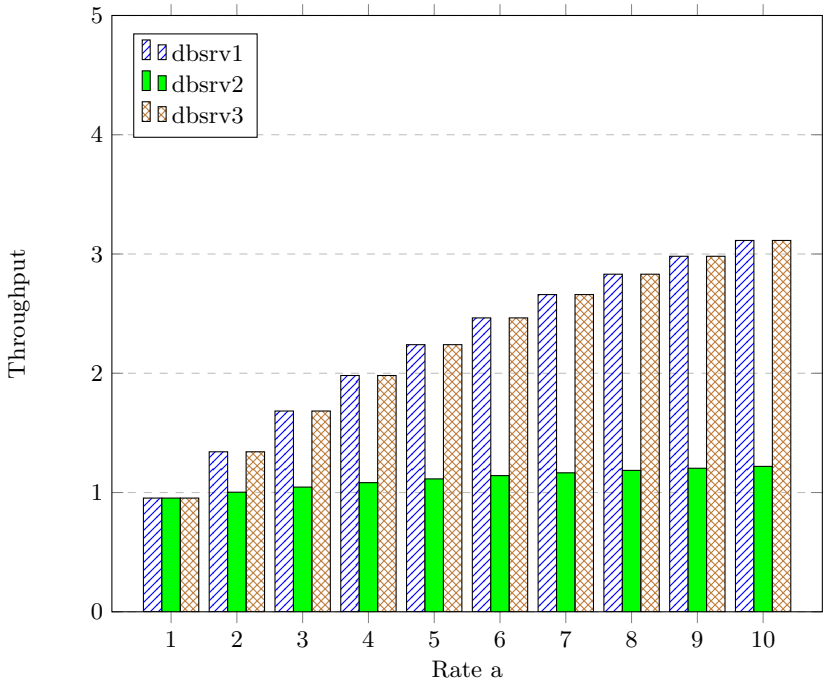


Fig. 13. Distributed database with replication - database throughput



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 14). The PEPA model of this (Figure 15) 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 16 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. 14. Simple microservices architecture

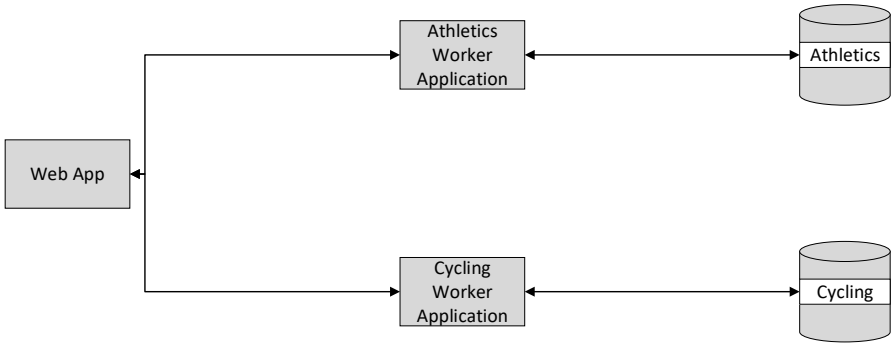


Fig. 15. Simple microservices PEPA model

$$
\begin{aligned}
a &= 1.0 - 10.0 \\
c &= 1.0 \\
w &= 100.0 \\
db &= 6.5
\end{aligned}
$$

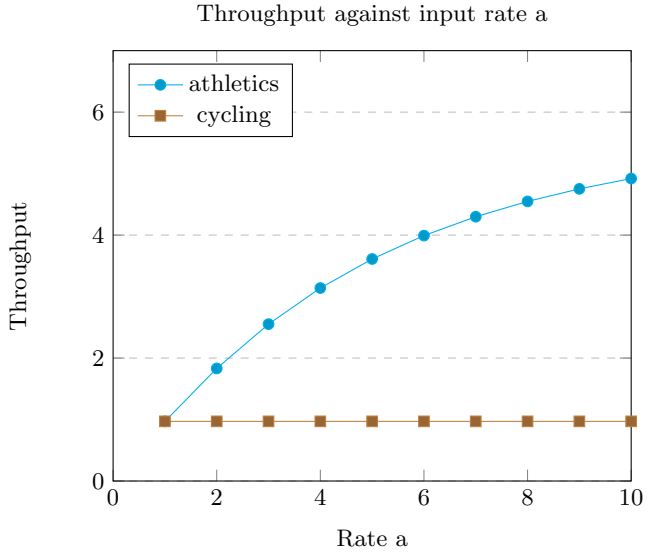
$$
\begin{aligned}
Website &\stackrel{def}{=} (athletics, a).Website + (cycling, c).Website \\
Worker_A &\stackrel{def}{=} (athletics, \top).WorkerSrv_A \\
WorkerSrv_A &\stackrel{def}{=} (workerA, \top).Worker_A \\
Worker_C &\stackrel{def}{=} (cycling, \top).WorkerSrv_C \\
WorkerSrv_C &\stackrel{def}{=}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

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

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. 16. Simple microservices experimental results



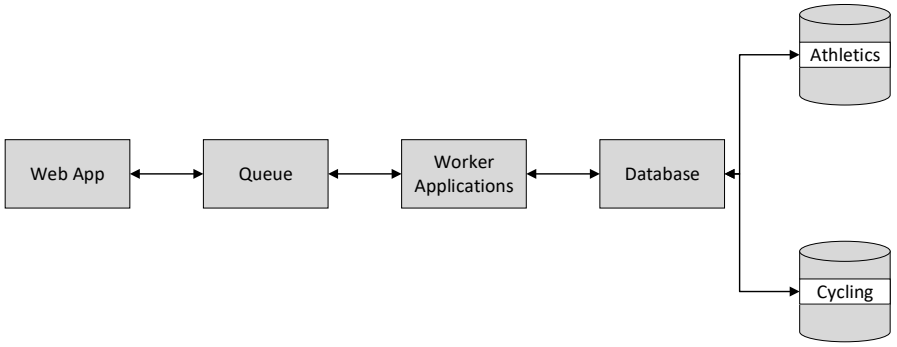
6.2 Shared queue and distributed database

The next system model (Figure 17) 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 18) 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. 17. 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 19. 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 model. This may indicate the decoupling 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

Fig. 18. Shared queue and distributed database

$$
\begin{aligned}
a &= 1.0 \\
c &= 1.0 \\
q &= 100.0 \\
db &= 5.0
\end{aligned}
$$

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

$$
\begin{aligned}
DB_1 &\stackrel{def}{=} (\mathit{queueA}, q).DBsrv_1 \\
DBsrv_1 &\stackrel{def}{=} (\mathit{dbsrv1}, db).DB_1 \\
DB_2 &\stackrel{def}{=} (\mathit{queueC}, q).DBsrv_2 \\
DBsrv_2 &\stackrel{def}{=} (\mathit{dbsrv2}, db).DB_2 \\
\mathit{Service}_1 &\stackrel{def}{=} (\mathit{dbsrv1}, db).\mathit{Service}_1 \\
\mathit{Service}_2 &\stackrel{def}{=} (\mathit{dbsrv2}, db).\mathit{Service}_2
\end{aligned}
$$

$$
\mathit{Website} \underset{\mathit{cycling}}{\overset{\mathit{athletics}}{\bowtie}} Q_0[10] \underset{\mathit{queueC}}{\overset{\mathit{queueA}}{\bowtie}} DB_1 \parallel DB_2 \underset{\mathit{dbsrv2}}{\overset{\mathit{dbsrv1}}{\bowtie}} \mathit{Service}_1 \parallel \mathit{Service}_2
$$

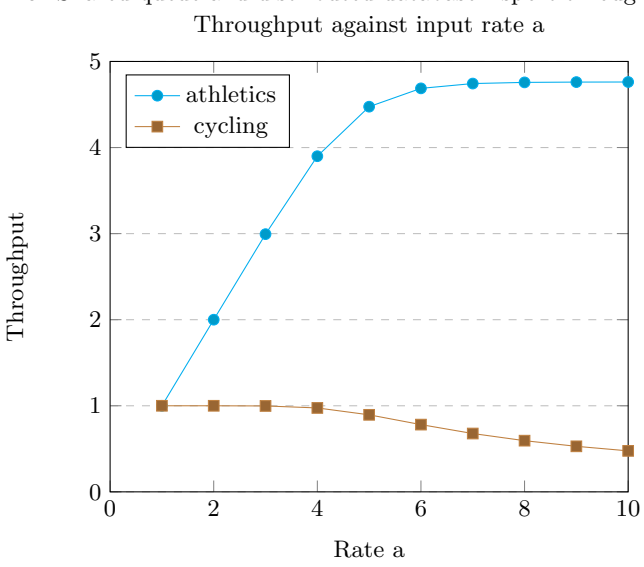
behaviour of the queue appears to be the most significant when combined into a system.

– the database node throughput follows the throughput of each sport activity, i.e. the partitioning strategy routes all of 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

Fig. 19. Shared queue and distributed database - sport throughput



6.3 Shared queue and distributed database with replication

The final system model (Figure 20) 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 21) 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. 20. Distributed database with replication architecture

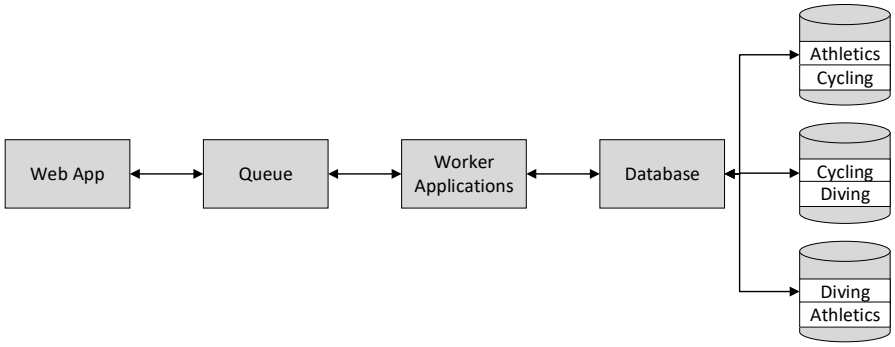


Fig. 21. Shared queue and distributed database with replication

$$
\begin{aligned}
a &= 1.0 \\
c &= 1.0 \\
d &= 1.0 \\
q &= 100.0 \\
db &= 5.0
\end{aligned}
$$

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

$$
\begin{aligned}
DB_1 &\stackrel{def}{=} (\mathit{queueA}, q).DBsrv_1 + (\mathit{queueC}, q).DBsrv_1 \\
DBsrv_1 &\stackrel{def}{=} (\mathit{dbsrv1}, \top).DB_1 \\
DB_2 &\stackrel{def}{=} (\mathit{queueC}, q).DBsrv_2 + (\mathit{queueD}, q).DBsrv_2 \\
DBsrv_2 &\stackrel{def}{=} (\mathit{dbsrv2}, \top).DB_2 \\
DB_3 &\stackrel{def}{=} (\mathit{queueD}, q).DBsrv_3 + (\mathit{queueA}, q).DBsrv_3 \\
DBsrv_3 &\stackrel{def}{=} (\mathit{dbsrv3}, \top).DB_3 \\
\mathit{Service}_1 &\stackrel{def}{=} (\mathit{dbsrv1}, db).\mathit{Service}_1 \\
\mathit{Service}_2 &\stackrel{def}{=} (\mathit{dbsrv2}, db).\mathit{Service}_2 \\
\mathit{Service}_3 &\stackrel{def}{=} (\mathit{dbsrv3}, db).\mathit{Service}_3
\end{aligned}
$$

$$
\mathit{Website} \underset{\substack{\mathit{athletics}\\ \mathit{cycling}\\ \mathit{diving}}}{\bowtie} Q_0[10] \underset{\substack{\mathit{queueA}\\ \mathit{queueC}\\ \mathit{queueD}}}{\bowtie} DB_1 \parallel DB_2 \parallel DB_3 \underset{\substack{\mathit{dbsrv1}\\ \mathit{dbsrv2}\\ \mathit{dbsrv3}}}{\bowtie} \mathit{Service}_1 \parallel \mathit{Service}_2 \parallel \mathit{Service}_3
$$

Experiments are performed in the Eclipse plugin with the usual input rates. The resulting steady state throughputs are shown in Table 6, in Figure 22 as a chart comparing *athletics* and *cycling* throughput (as the cycling and diving throughputs are identical), and in Figure 29 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

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

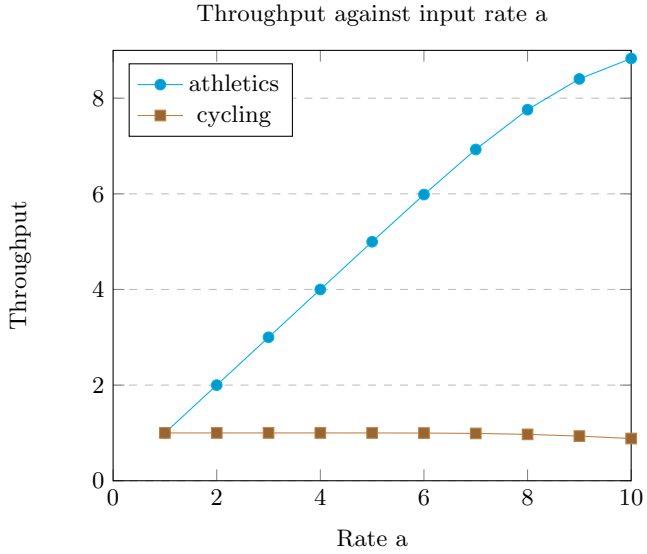
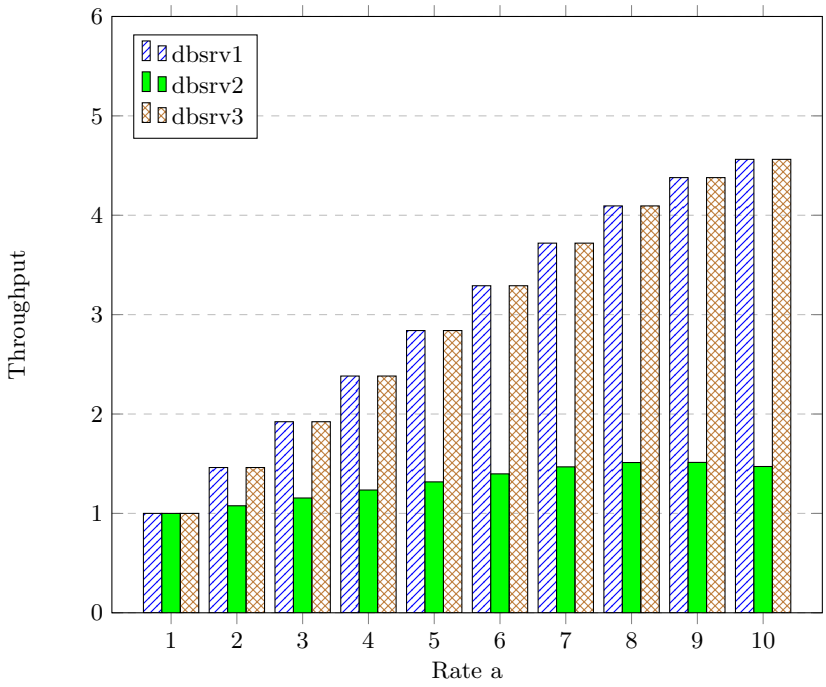


Fig. 23. Shared queue and distributed database with replication - database throughput



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 numerical results of each system model test are compared in Table 7. Figure 24 shows how the athletics and cycling throughputs behave for each system in graphical form. 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.

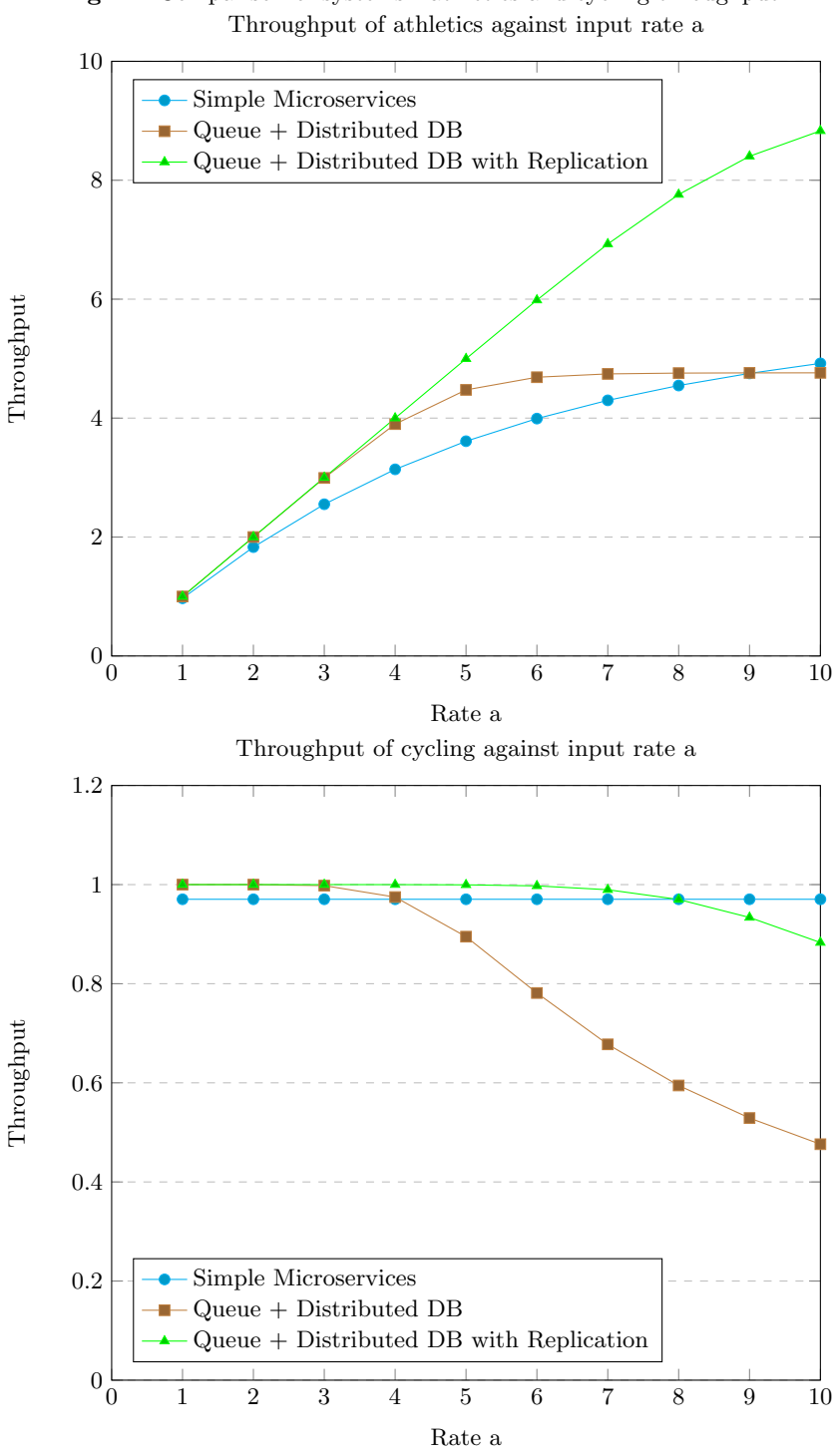
Table 7. Comparison of system results

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 adds decoupling, resulting in 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.

Fig. 24. Comparison of systems - athletics and cycling throughput



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 specifics of each design are discussed by system, but the common feature is that all three systems use a Cassandra database or databases with the same ticket schema design, given in Figure 25, 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].

Fig. 25. Cassandra database ticket schema

```
/*
* 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';
```

Workload Simulation. The workload from a web application and its users was simulated using Apache JMeter [4], a Java application designed 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. The required demand may be increased by increasing the number of threads used in the test. 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

Two completely separate Cassandra databases, each on an Azure Standard F1s (1 core, 2 GB memory) Ubuntu Virtual Machine, one containing Athletics tickets, one containing Cycling tickets.

Two worker applications using RESTful APIs using Java Spring [24]. Each worker application runs on a separate Azure Standard DS1 v2 (1 core, 3.5 GB memory) Ubuntu Virtual Machine. Each connects to one of the Cassandra databases.

Each worker application has a
control API which doesn't access the database, but for which metrics are recorded. Each worker application also has a
search API which takes a sport parameter (Athletics or Cycling) and queries the database for all matching tickets.

Use JMeter with Poisson random timer (negative exponential distribution) with a lambda value of 500 milliseconds, Cycling at a constant 10 threads/users (approximately 20 requests per second) and Athletics ramping up from 10-100 in steps of 10, so the desired demand is 20-200 requests per second. Each thread group has a loop count of 500 requests, ensuring several minutes worth of samples and therefore a number of rolling 1-minute averages.

The control version of the above JMeter test plan uses the same variables, but sends both Athletics and Cycling requests to the
control API.

Limit of throughput using
search was measured at 130 queries per second. This is lower than the throughput suggested by using the cassandra-stress tool, and suggests that using Java Spring Data adds overheads to the database requests (most likely, as the RESTful requests to the worker applications are sessionless, this is starting a new database session for each request).

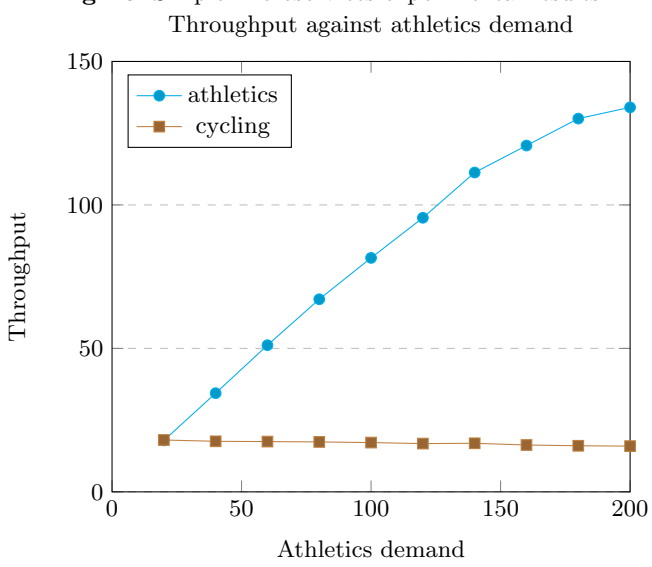
See the experimental results in Table 8.

Control shows that throughput approaches demand (difference likely to be due to random distribution, network latency, etc). However the Athletics demand is throttled by the database throughput. The Cycling throughput is unaffected by the Athletics demand.

Table 8. Simple microservices experimental results

Athletics				Cycling			
users	rate	search	control	users	rate	search	control
10	20	17.92	19.286	10	20	18.076	19.252
20	40	34.386	37.672	10	20	17.614	18.992
30	60	51.114	56.442	10	20	17.522	18.962
40	80	67.132	74.61	10	20	17.402	18.754
50	100	81.54	92.586	10	20	17.176	18.732
60	120	95.518	111.598	10	20	16.824	18.7
70	140	111.298	131.43	10	20	16.928	18.83
80	160	120.698	150.494	10	20	16.326	18.852
90	180	130.088	168.29	10	20	16.062	18.826
100	200	134.01	185.846	10	20	15.936	18.718

Fig. 26. Simple microservices experimental results



7.3 Shared queue middleware

This would normally be used for the return ticket operation. To ensure that a usable Cassandra metric was available, a search operation was used again.

Distributed Cassandra databases using two nodes each on an Azure Standard F1s (1 core, 2 GB memory) Ubuntu Virtual Machine, using a Distributed keyspace with SimpleStrategy, replication_factor=1. Cassandra's partitioning places Athletics tickets on one node, and Cycling tickets on the other. This is validated using nodetool getendpoints e.g.

nodetool getendpoints Distributed ticket Athletics

Cassandra configured to record metrics of a count of all completed read queries every 10s i.e.

org.apache.cassandra.metrics.ThreadPools.CompletedTasks.request.ReadStage

A Python script calculates the rolling 1-minute average throughput from these counts.

A single shared Azure Storage Queue is used.

A single multithreaded QueueWorker application dequeues every request from the shared Azure Storage Queue. It runs on an Azure Standard DS3 v2 Promo (4 cores, 14 GB memory) Ubuntu Virtual Machine. As populating the queue with JMeter and processing it with the 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 tech-

nique suggested that maximum performance came with QueueWorker running with 16 threads.

Note that without the overheads of starting a new Cassandra database session for each request, it is necessary to slow Cassandra down by turning on tracing for 100% of queries - bin/nodetool settraceprobability 1.0

QueueWorker processes both Control tickets and real (Athletics, Cycling) tickets. Metrics are recorded for all requests, but for real ticket requests a database select of all tickets for the matching sport is carried out first and the metric is only recorded if the query returns results.

Use JMeter with Poisson random timer (negative exponential distribution) with a lambda value of 100 milliseconds, Cycling at a constant 15 threads/users (approximately 95 requests per second) and Athletics ramping up from 15-150 in steps of 15, so the desired demand is 95-950 requests per second. Each thread group has a loop count of 1500 requests, ensuring several minutes worth of samples and therefore a number of rolling 1-minute averages.

The control version of the above JMeter test plan uses the same variables, but sending Control tickets rather than Athletics and Cycling tickets to the queue.

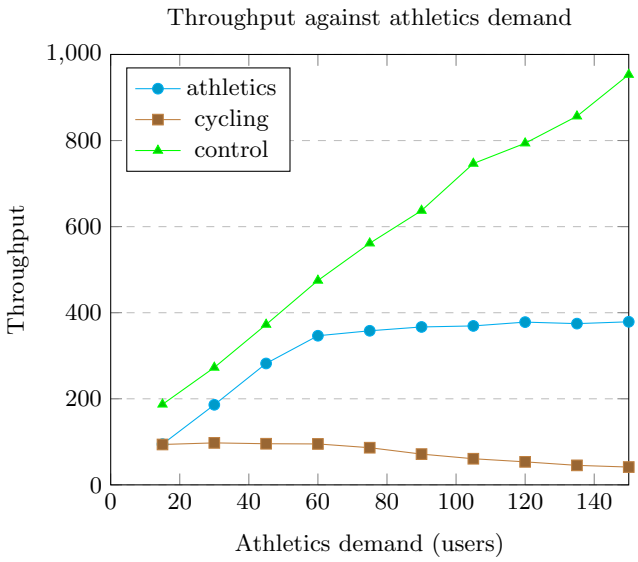
Does the predicted result, that the queue will be the overriding constraint, still hold true with a real, effectively unlimited queue?

See the experimental results in Table 9.

Table 9. Shared queue with distributed DB experimental results

Control		Athletics		Cycling		Database		
users	rate	users	rate	users	rate	db1	db2	ratio
30	187.028	15	94.554	15	93.832	97.882	98.964	1.01
45	272.694	30	186.024	15	97.528	101.464	194.228	1.91
60	372.476	45	282.04	15	95.562	97.664	293.268	2.95
75	474.85	60	346.472	15	95.082	96.13	362.94	3.64
90	561.374	75	358.004	15	86.33	84.96	365.02	4.15
105	637.288	90	366.896	15	71.462	72.04	373.196	5.13
120	746.532	105	369.286	15	60.704	61.386	376.626	6.08
135	793.756	120	378.138	15	53.468	53.934	385.616	7.07
150	856.462	135	374.624	15	45.306	45.514	381.08	8.27
165	953.174	150	378.9	15	41.38	41.9	388.406	9.16

Fig. 27. Shared queue with distributed DB experimental results



7.4 Distributed database with replication

Distributed Cassandra databases using three nodes each on an Azure Standard F1s (1 core, 2 GB memory) Ubuntu Virtual Machine, using a Replicated keyspace with SimpleStrategy, replication_factor=2. Cassandra's partitioning places Athletics, Cycling and Diving tickets on different nodes with each node also containing replicas of one other ticket type. Note that it was necessary to use ByteOrderedPartitioner to force this.

This is validated using nodetool getendpoints e.g.

nodetool getendpoints Distributed ticket Athletics

Athletics/Cycling Cycling/Diving Diving/Athletics

Cassandra metrics configured and processed as above.

The same shared Azure Storage Queue and QueueWorker application running with 16 threads used again.

Cassandra with tracing for 100% of queries as before.

Use JMeter with Poisson random timer (negative exponential distribution) with a lambda value of 100 milliseconds, Cycling and Diving at a constant 15 threads/users (approximately 95 requests per second) and Athletics ramping up from 15-150 in steps of 15, so the desired demand is 95-950 requests per second. Each thread group has a loop count of 1500 requests, ensuring several minutes worth of samples and therefore a number of rolling 1-minute averages.

The control version of the above JMeter test plan uses the same variables, but sending Control tickets rather than real tickets to the queue.

See the experimental results in Table 10.

Table 10. 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

Fig. 28. Shared queue and distributed database with replication - sport throughput

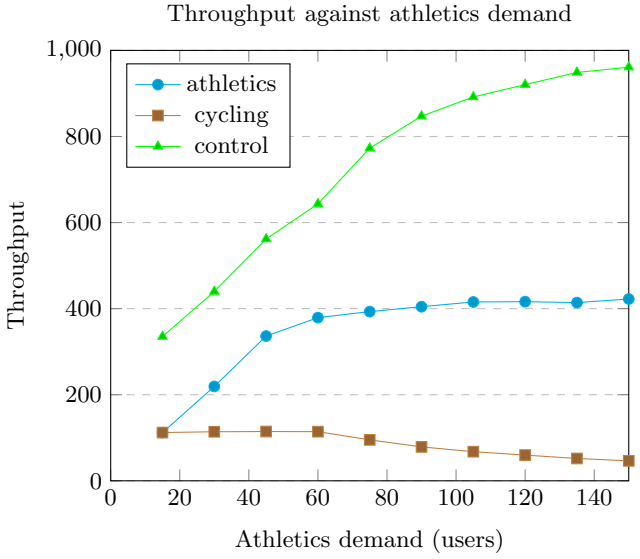
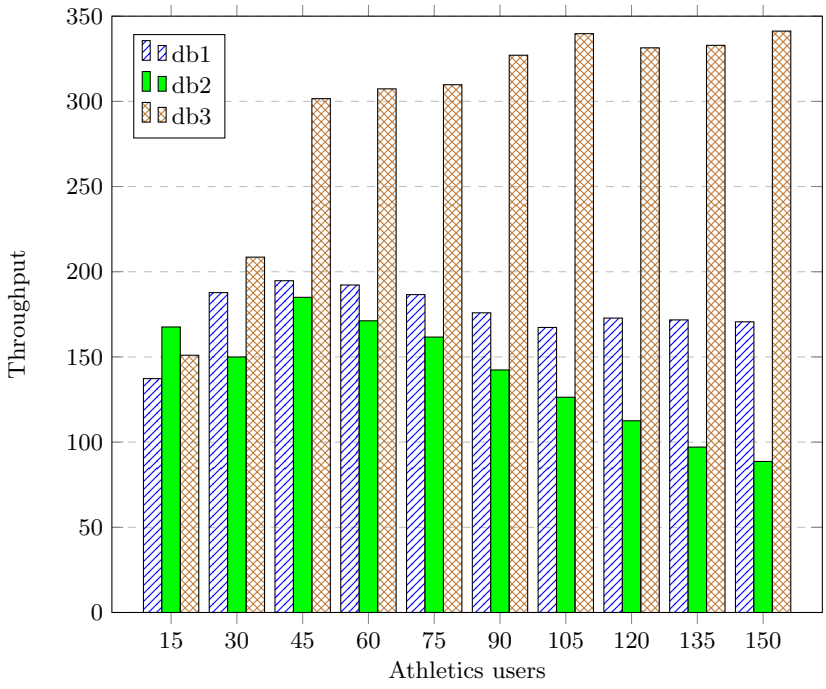


Fig. 29. Shared queue and distributed database with replication - database throughput



8 Conclusions

The models are useful for qualitative prediction. They successfully predicted the non-trivial result that when a shared queue is used in combination with a distributed database, whether or not replication is used, then once the partitions storing the high demand tickets were no longer able to satisfy it, the throughput of the other resources was choked in proportion to the relative demand between them and the skewed resources. The models also predicted that when using replication, there would also be throughput at the replica node.

However, the models were less successful at quantitative predictions. When using replication, the throughput was not spread evenly (or randomly) between database nodes, and this also meant that the system was not able to satisfy as much demand as predicted. This means that it was not possible to use the models to compare which system would make best use of the resources available, as the models suggested - i.e. the models as they stand are not suitable for right-sizing.

Conclusions about the modelled/built systems - how well they isolate demand vs how well they utilise resources, whether or not a shared queue worked, whether or not the database partitioning strategy worked

9 Future Work

With a set of simple models, there is much potential for future work to build on them.

The real Cassandra database behaviour is more complex than described by these simple models. An area of future work might be to adapt these models to make them closer to the true behaviour. However models tied closely to a particular database implementation are likely to be less universally applicable.

Other areas - different partitioning strategies. For example, where the partitioning strategy does not attempt to isolate demand, but shares it between database nodes,

Different queue strategies. A priority queue for the skewed demand? Choking skewed demand in the queue before it overwhelms the whole database? Trigger based on when the normal demand resources begin to get affected by the skewed demand.

An interesting area of future work might be in using the modelling techniques in adaptive algorithms.

Models might be used to change the queueing policy on-demand.

A model might be used as a policy for elastic scaling? and compared with the performance of other right-sizing strategies; control theory, machine learning and other model based techniques including statistical.

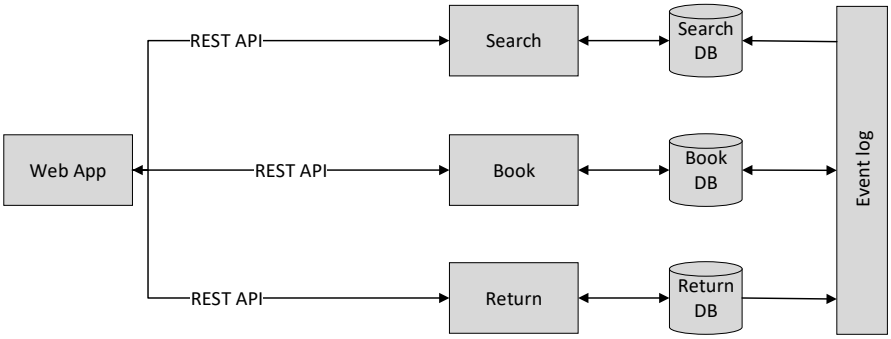
Data partitioning - where the high demand is unknown in advance, we need an adaptive strategy. Workload-aware clustering algorithms do exist for the

placement of new data, e.g. [17], but our use case has a fixed set of tickets. Re-placement of existing data onto different partitions would be likely to require many reads, writes and deletes.

A more 'natural' microservices architecture that partitions the system by operation (Book, Search, Return) with a separate database for each. The databases maintain eventual consistency via an event streaming application e.g. using Kafka. A new model for this, it may be generalised to a model of publish/-subscribe message oriented middleware.

1. Book is an event producer and consumer (produces when a ticket is booked, consumes returned tickets).
2. Search is an event consumer (consumes the state of tickets that are booked and returned).
3. Return is an event producer (produces returned tickets).

Fig. 30. Operational microservices architecture



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. Kamal, J., Murshed, M., Buyya, R.: Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable oltp applications. Future Generation Computer Systems 56, 421–435 (2016)
18. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)

19. Lewis, J., Fowler, M.: Microservices (2014), martinfowler.com, [Online; accessed 5-March-2017]
20. 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
21. Mell, P., Grance, T.: The nist definition of cloud computing (2011)
22. Microsoft: Azure storage documentation (2017), https://docs.microsoft.com/en-us/azure/storage/, [Online; accessed 27-July-2017]
23. 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]
24. Pivotal: Spring (2017), http://spring.io/, [Online; accessed 28-June-2017]
25. 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]
26. 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]
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)