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. In this article we present a ticketing use case and argue that at each layer of the architecture, the distributed computing technologies of the Cloud may maintain throughput to the lower demand resources, maximising the available functionality of the system.

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 [20]. HBO Go was brought down by demand for the finale of "True Detective" [9]. Apple's iTunes Store suffered outage on the launch day of the iPhone 7 (new iPhone registration is carried out via an iTunes function) [25].

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 built systems.

2 Background

Consider a general OLTP application using a distributed architecture. 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.

Adapting. A system using Cloud technologies may adapt to increased demand. *Rapid elasticity* is an essential characteristic of Cloud Computing by the NIST definition [17]. 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 [10]. 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 skewed demand for *athletics* tickets.

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

Some notes about the choice of technologies... two aspects, right-sizing and measuring/routing demand and throughput, have chosen the latter, refer to some of the work on the former?

3.1 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. Also, failure of one of the processes in a distributed system may cause failure of the system as a whole.

Synchronous vs Asynchronous Middleware. 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. The system components are tightly coupled. This is undesirable for our ticketing application.

Distributed systems using some form of 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.

The "return" operation use case 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.

Message-Oriented Middleware (MOM). MOM is a form of Asynchronous Middleware, commonly provided by Larger Cloud service providers such as Amazon Web Services and Microsoft Azure. These brokered message services provide 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 [8].

There are two main messaging models, both of which are offered by Microsoft Azure Service Bus [18] for example.

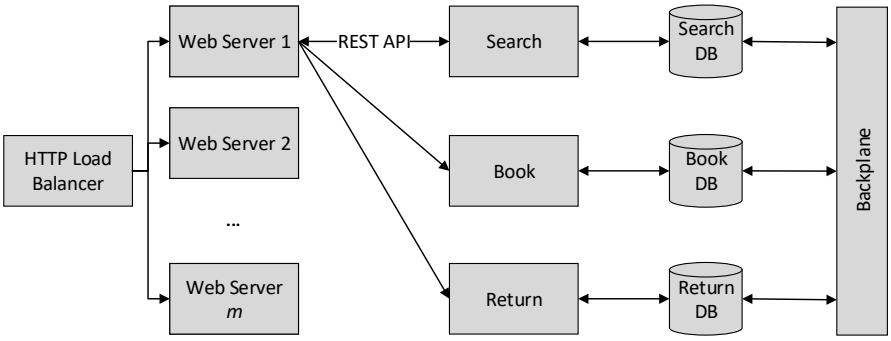
Point-to-Point Queues. Azure Queues are a point-to-point service implementing First In, First Out (FIFO) message delivery. Many processes may send messages to a queue, and each message is received by one consumer - though it may be one of several consumers competing for messages from this queue. This competing consumer pattern offers a means of balancing load from the Web servers between the Worker Applications in the ticketing use case.

Publish/subscribe. Publish/subscribe (in Azure, topics and subscriptions) are a properly one-to-many or many-to-many communication mechanism. Any single producer may send one message to a topic, and then all consumers that subscribe to that topic receive a copy of the same message.

3.2 Microservices

Microservice architecture is an approach to structuring applications as suites of small services, defined by business capability verticals rather than technological layers [16] [22]. 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 [23]. 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. 1. Microservices



3.3 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 [6]. 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 [1].

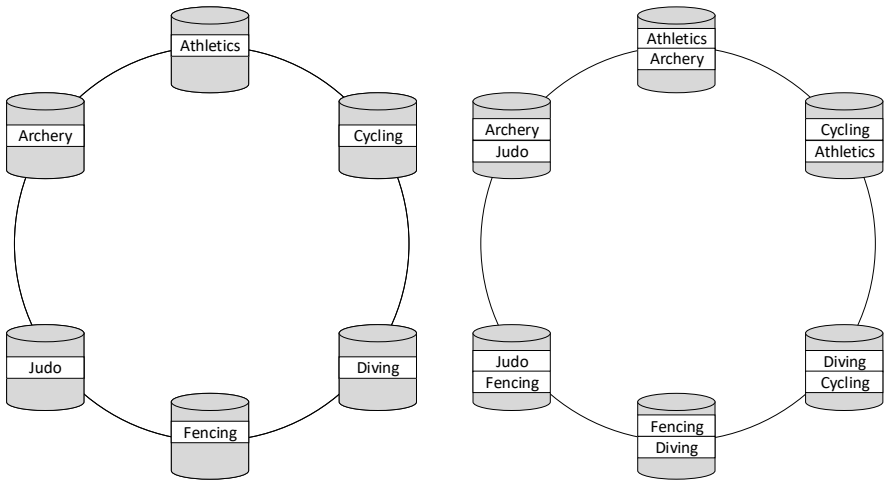
In the use case, the quantity of data does not approach the levels of "Big Data" applications. Partitioning is proposed instead as a means of scaling the

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.

Discuss Cassandra replication model, and refer to other distributed databases using it

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 system may failover to a copy of the data on another data node, but this will just transfer the demand elsewhere. If this is also the primary data node of an otherwise low demand data type, then it may be overwhelmed in turn.

Fig. 2. Distributed database, without and with replication



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 [5] 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 [13]) model throughput in interdependent processes, with a mixture of independent and shared actions operating at different rates. There is a PEPA Workbench tool [12] 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 *activites* 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 [11]:

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, where the number of instances in each state is important but the individual states are not signficant.

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 PEPA workbench [12] 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 for any behavioural insights.

5.1 Shared middleware queue

Queueing systems have already been extensively modelled in PEPA [26].
Queue shared between two processes.
Notes on applying this work to model a real Azure Storage Queue. A real queue is effectively unlimited, determine what smaller queue size we can use for practicality (smaller state space).
The PEPA model for a general shared queue is shown in Figure 3.

Fig. 3. Shared queue PEPA model

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

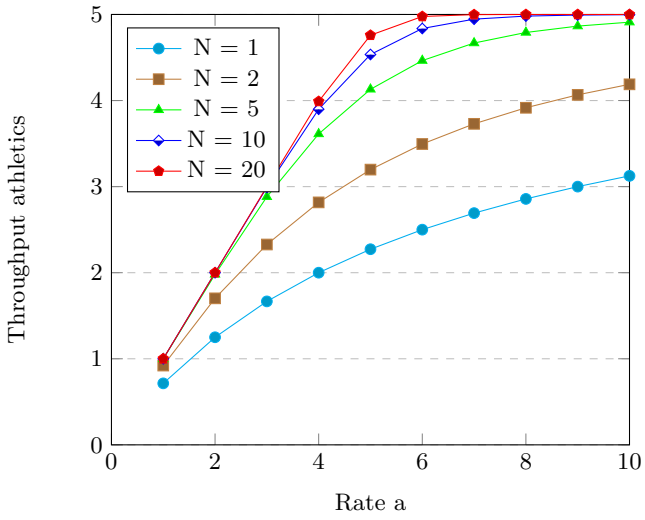
$$
\begin{aligned}
Arrival_A &\stackrel{def}{=} (athletics, a).Arrival_A \\
Arrival_C &\stackrel{def}{=} (cycling, c).Arrival_C \\
Service_1 &\stackrel{def}{=} (serve1, s1).Service_1 \\
Service_2 &\stackrel{def}{=} (serve2, s2).Service_2 \\
Q_0 &\stackrel{def}{=} (athletics, \top).Q_1 + (cycling, \top).Q_2 \\
Q_1 &\stackrel{def}{=} (serve1, \top).Q_0 \\
Q_2 &\stackrel{def}{=} (serve2, \top).Q_0
\end{aligned}
$$

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

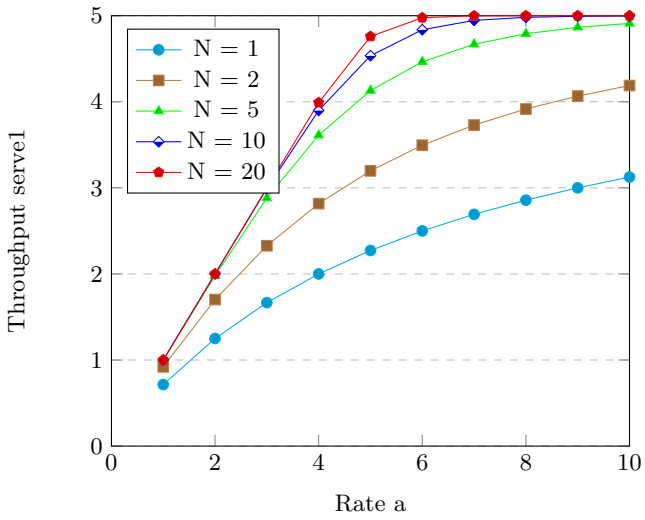
5.2 Database models

Fig. 4. Shared queue experimental results

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



Throughput of serve1 against input rate a for different queue lengths N



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

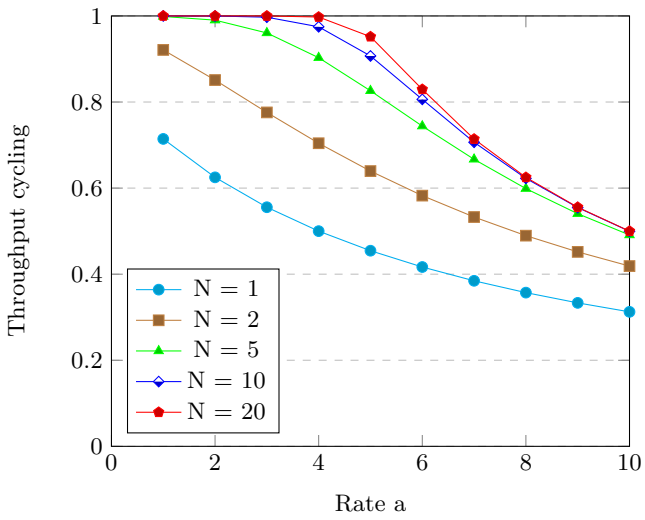


Table 1. Shared queue N=10 experimental results

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

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, e.g.

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

What doesn't this model? Statement about how the models below show that it can still be a useful building block.

Distributed database. Figure 5 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. Both nodes execute in parallel without cooperating on any activities. Experiments are carried out in PEPA workbench 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 database node to service. The results appear in Table 2.

Fig. 5. Distributed database PEPA model

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

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

$$
\mathit{Website} \underset{\substack{\mathit{athletics}\\ \mathit{cycling}}}{\bowtie} \mathit{DB}_1 \parallel \mathit{DB}_2 \underset{\substack{\mathit{dbsrv1}\\ \mathit{dbsrv2}}}{\bowtie} \mathit{Service}_1 \parallel \mathit{Service}_2
$$

Table 2. Distributed database 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. The PEPA model for a distributed database is shown in Figure 7.

See the experimental results in Table 3.

Fig. 6. Distributed database experimental results

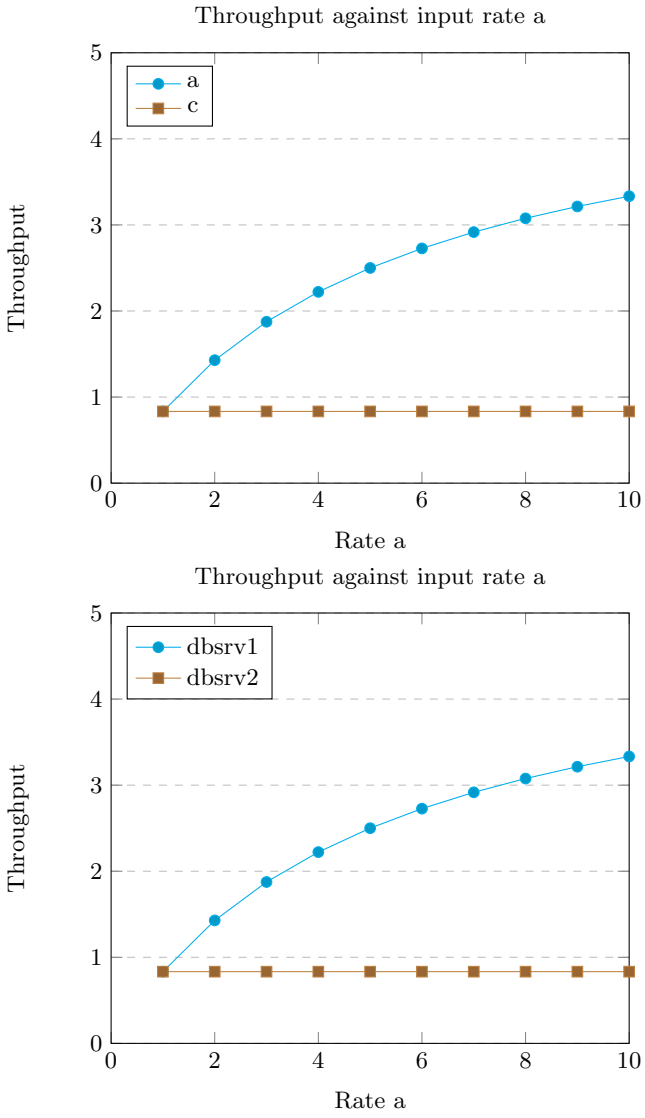


Fig. 7. Distributed database with replication PEPA model

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

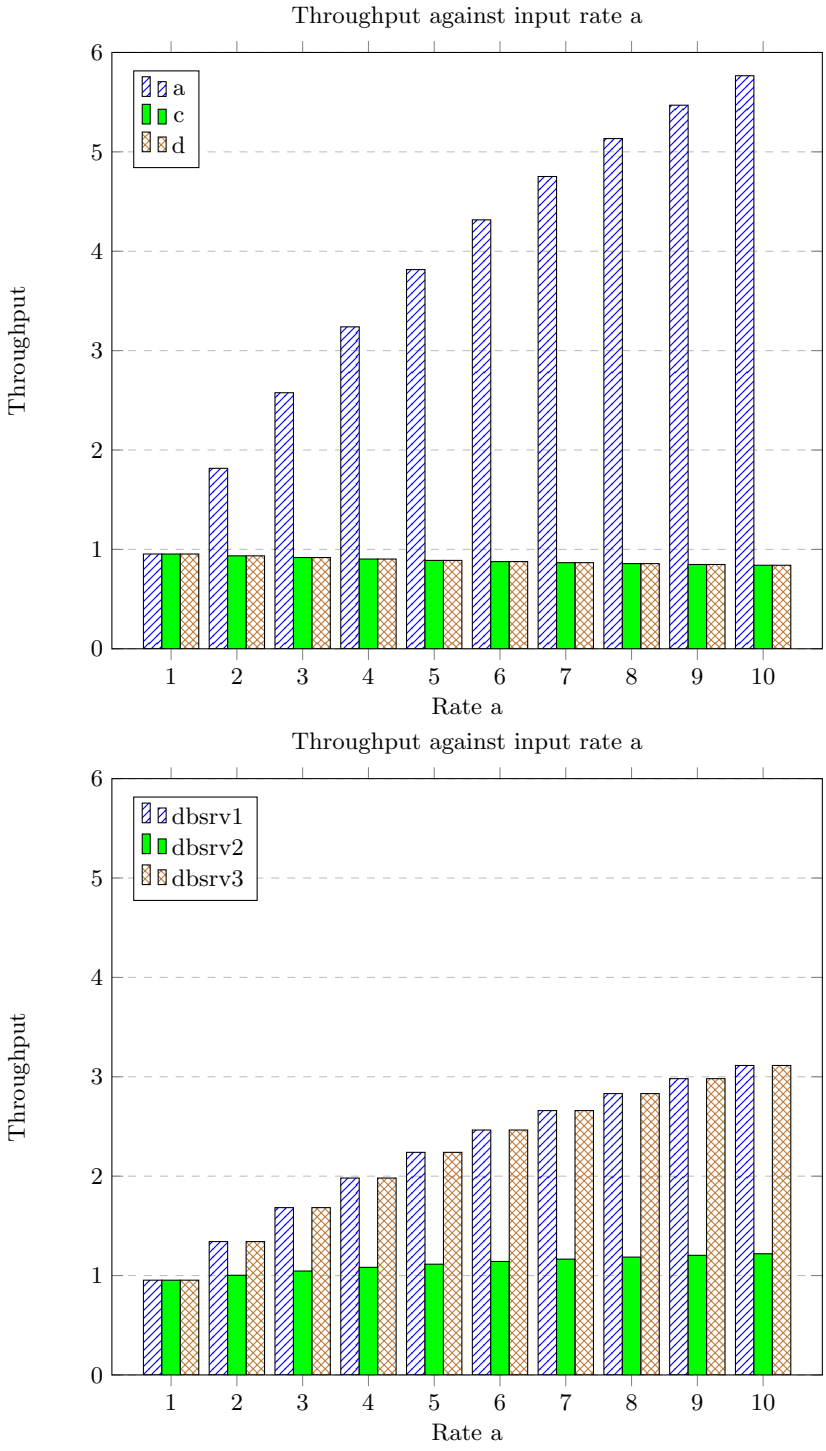
$$
\begin{aligned}
Website &\stackrel{def}{=} (athletics, a).Website + (cycling, c).Website + (diving, d).Website \\
DB_1 &\stackrel{def}{=} (athletics, \top).DBsrv_1 + (cycling, \top).DBsrv_1 \\
DBsrv_1 &\stackrel{def}{=} (dbsrv1, \top).DB_1 \\
DB_2 &\stackrel{def}{=} (cycling, \top).DBsrv_2 + (diving,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

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

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. 8. Distributed database with replication experimental results



6 PEPA System Models

The proposed system will use distributed architectures. Users will access it from a web-based front end. Tickets will be stored in a database partitioned across several data nodes. In between the web servers and database will be a number of worker applications to service user requests, connected to the web servers by some middleware.

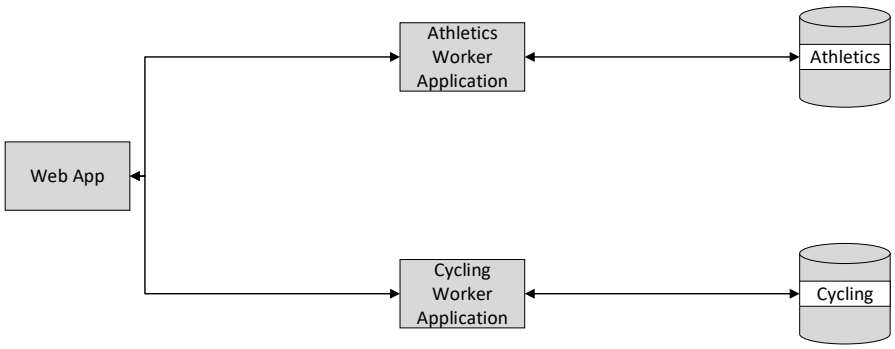
For each architecture it will be assumed that the web application will be designed to cope with the required demand, using a cluster of web servers where the throughput is managed using some HTTP Load Balancing algorithm [10], and potentially Elastic Scaling of servers e.g. using the autoscaling features of Amazon Web Services [2] or Microsoft Azure [19].

6.1 Simple microservices

There are two separate databases, one for Athletics tickets, one for Cycling.

It's expected that this architecture will lead to isolation of the skewed demand and that the results of testing the model will not be surprising, but that this will provide a useful control for other architectures.

Fig. 9. Simple microservices architecture



See the experimental results in Table 4.

6.2 Shared queue and distributed database

Requests via a shared queue to worker applications going to a distributed database with two nodes, Athletics and Cycling.

See the experimental results in Table 5.

Fig. 10. Simple microservices PEPA model

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

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

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

Fig. 11. Simple microservices experimental results

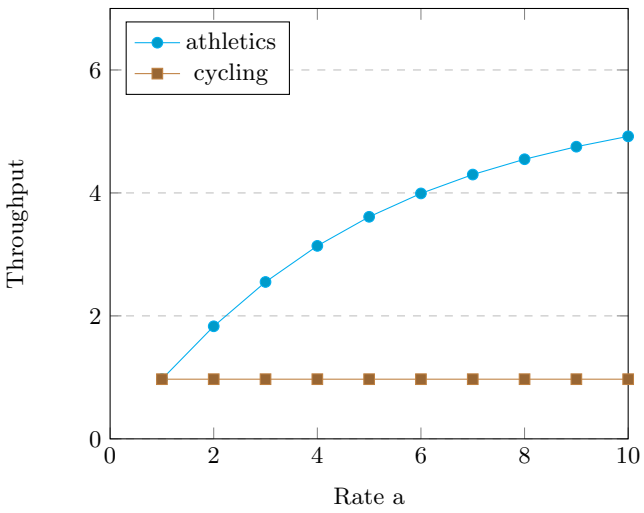


Fig. 12. Shared queue middleware architecture

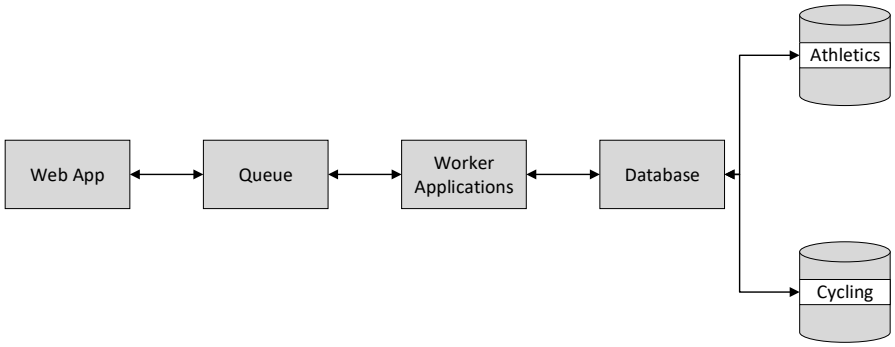


Fig. 13. Shared queue and distributed database

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

$$
\begin{aligned}
\mathit{Website} &\stackrel{def}{=} (\mathit{athletics}, a).\mathit{Website} + (\mathit{cycling}, c).\mathit{Website} \\
Q_0 &\stackrel{def}{=} (\mathit{athletics}, \top).Q_A + (\mathit{cycling}, \top).Q_C \\
Q_A &\stackrel{def}{=} (\mathit{queueA}, \top).Q_0 \\
Q_C &\stackrel{def}{=} (\mathit{queueC}, \top).Q_0 \\
DB_1 &\stackrel{def}{=} (\mathit{queueA}, q).DBsrv_1 \\
DBsrv_1 &\stackrel{def}{=} (\mathit{dbsrv1}, db).DB_1 \\
DB_2 &\stackrel{def}{=} (\mathit{queueC}, q).DBsrv_2 \\
DBsrv_2 &\stackrel{def}{=} (\mathit{dbsrv2}, db).DB_2 \\
\mathit{Service}_1 &\stackrel{def}{=} (\mathit{dbsrv1}, db).\mathit{Service}_1 \\
\mathit{Service}_2 &\stackrel{def}{=} (\mathit{dbsrv2}, db).\mathit{Service}_2
\end{aligned}
$$

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

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

Table 5. Shared queue and distributed database experimental results

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

6.3 Shared queue and distributed database with replication

Requests via a shared queue to worker applications going to a distributed database with three nodes, Athletics, Cycling and Diving, where each partition is replicated on another node.

See the experimental results in Table 6.

6.4 Comparison

The system results are compared in Table 7.

This shows that the simple microservices system does a good job of isolating the skewed demand from the rest of the system, but it is an inefficient use of the database resources. The actual throughput of the athletics demand is limited to its database's throughput, while the spare capacity of the cycling database

Fig. 14. Distributed database with replication architecture

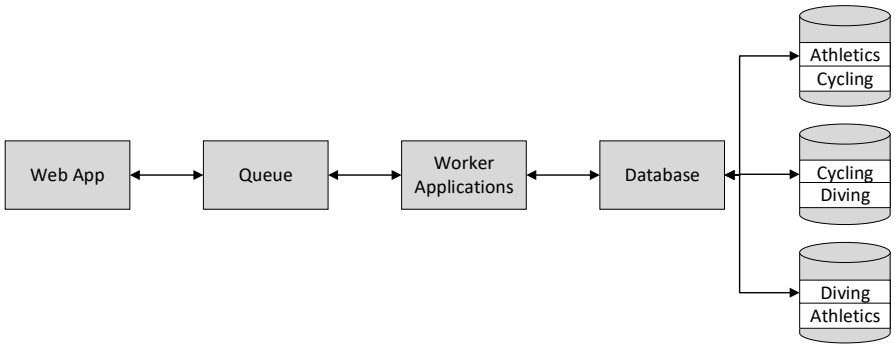


Fig. 15. Shared queue and distributed database with replication

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

$$
\begin{aligned}
\mathit{Website} &\stackrel{def}{=} (\mathit{athletics}, a).\mathit{Website} + (\mathit{cycling}, c).\mathit{Website} + (\mathit{diving}, d).\mathit{Website} \\
Q_0 &\stackrel{def}{=} (\mathit{athletics}, \top).Q_A + (\mathit{cycling}, \top).Q_C + (\mathit{diving}, \top).Q_D \\
Q_A &\stackrel{def}{=} (\mathit{queueA}, \top).Q_0 \\
Q_C &\stackrel{def}{=} (\mathit{queueC}, \top).Q_0 \\
Q_D &\stackrel{def}{=} (\mathit{queueD}, \top).Q_0 \\
DB_1 &\stackrel{def}{=} (\mathit{queueA}, q).DBsrv_1 + (\mathit{queueC}, q).DBsrv_1 \\
DBsrv_1 &\stackrel{def}{=} (\mathit{dbsrv1}, \top).DB_1 \\
DB_2 &\stackrel{def}{=} (\mathit{queueC}, q).DBsrv_2 + (\mathit{queueD}, q).DBsrv_2 \\
DBsrv_2 &\stackrel{def}{=} (\mathit{dbsrv2}, \top).DB_2 \\
DB_3 &\stackrel{def}{=} (\mathit{queueD}, q).DBsrv_3 + (\mathit{queueA}, q).DBsrv_3 \\
DBsrv_3 &\stackrel{def}{=} (\mathit{dbsrv3}, \top).DB_3 \\
\mathit{Service}_1 &\stackrel{def}{=} (\mathit{dbsrv1}, db).\mathit{Service}_1 \\
\mathit{Service}_2 &\stackrel{def}{=} (\mathit{dbsrv2}, db).\mathit{Service}_2 \\
\mathit{Service}_3 &\stackrel{def}{=} (\mathit{dbsrv3}, db).\mathit{Service}_3
\end{aligned}
$$

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

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

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

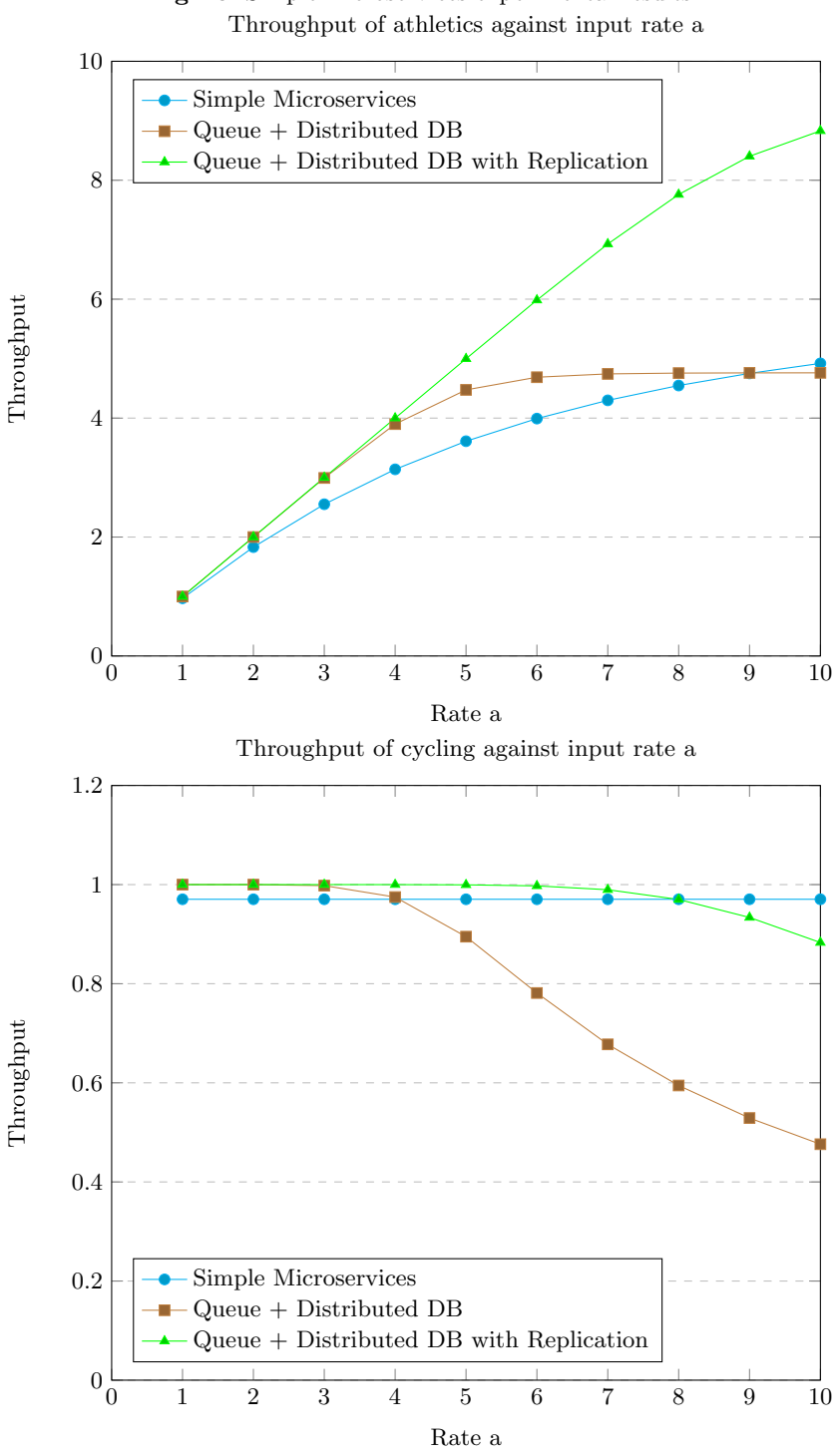
goes unused. Using a distributed database with replication by contrast uses the capacity of two database nodes to serve the skewed demand, so that the actual throughput is much closer to the desired value.

(NOTE: the replication model uses 3 nodes, the others use 2 - need to compare like with like. Try all with 3 or replication with 2?)

Table 7. Comparison of system results

Rate	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

Fig. 16. Simple microservices experimental results



7 Built systems

Reference to github at [24]
General design decisions:
Cassandra [15][3] database. There are 500 tickets of each sport stored in the databases in different configurations.
/* * 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 ticket owner (for booked ticket) * * The partition key is sport * The clustering columns are owner, day, id */
The web application and its users were simulated using Apache JMeter [4] to consume the RESTful APIs of each system (for the simple microservices architecture, the Java Spring APIs; for the shared queue architectures, the Microsoft Azure Storage Queue REST APIs).

Measurands. Throughputs as described below.

Measurement method. Measurement using Meters from Coda Hale Metrics [7]. Metrics are logged every 10 seconds.

Measurement procedure. The largest 1-minute moving average over a run for a chosen demand is extracted by a Python script. Each experiment was carried out 5 times and the mean was taken of the 5 sets of results.

7.1 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 [21]. 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

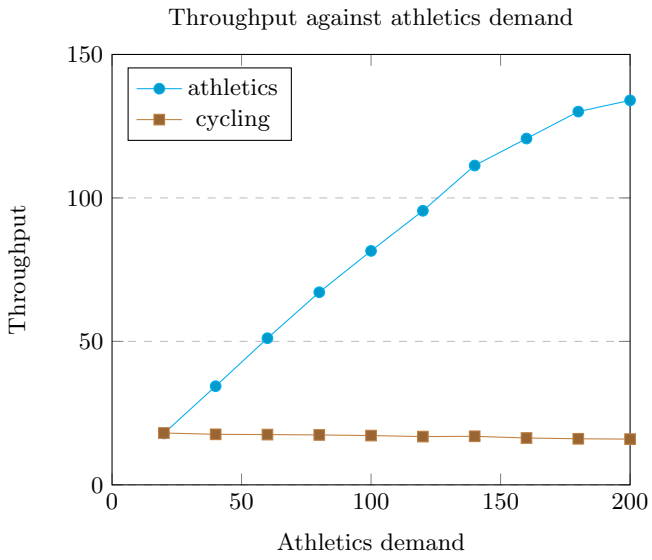
Athletics				Cycling			
users	rate	search	control	users	rate	search	control
10	20	17.92	19.286	10	20	18.076	19.252
20	40	34.386	37.672	10	20	17.614	18.992
30	60	51.114	56.442	10	20	17.522	18.962
40	80	67.132	74.61	10	20	17.402	18.754
50	100	81.54	92.586	10	20	17.176	18.732
60	120	95.518	111.598	10	20	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

7.2 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

Fig. 17. Simple microservices experimental results



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 technique 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

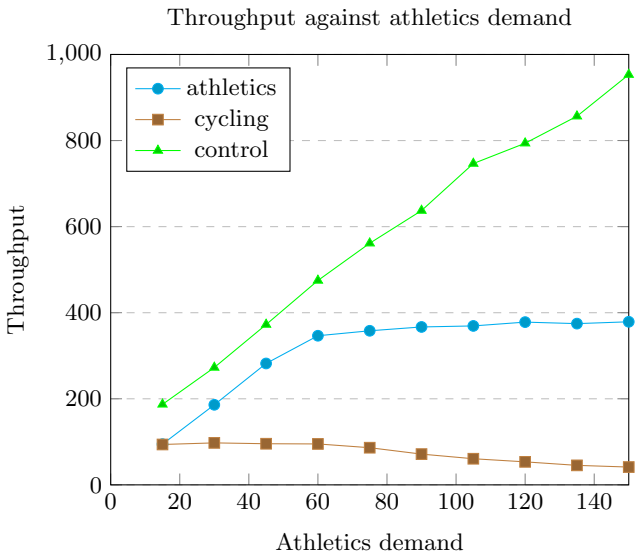
7.3 Distributed database with replication

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

Fig. 18. Shared queue with distributed DB experimental results



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 9.

8 Conclusion and Future Work

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.

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.

Different queue strategies. A priority queue for the skewed demand?

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

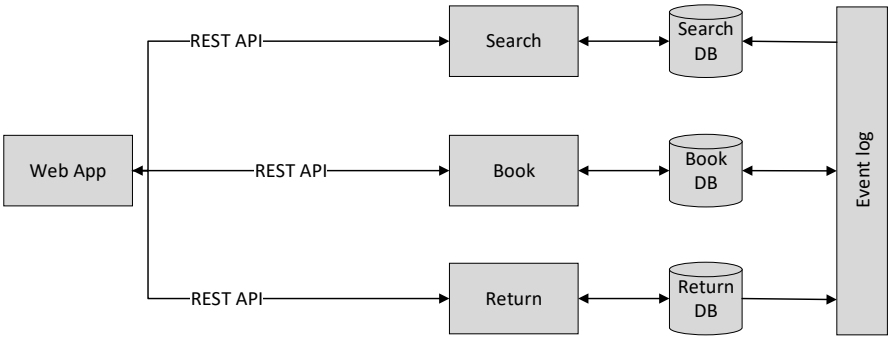
Data partionining - where the high demand is unknown in advance, we need an adaptive strategy. Workload-aware clustering algorithms do exist for the placement of new data, e.g. [14], but our use case has a fixed set of tickets. Re-placement of existing data onto different partitions would be likely to require many reads, writes and deletes.

8.1 Operational microservices

A more ‘natural’ microservices architecture 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.

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. 19. Operational microservices architecture



References

1. 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)
2. Amazon Web Services Inc: Auto scaling (2017), https://aws.amazon.com/autoscaling/, [Online; accessed 5-March-2017]
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. 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)
6. Cattell, R.: Scalable sql and nosql data stores. Acm Sigmod Record 39(4), 12–27 (2011)
7. Coda Hale: Metrics (2014), http://metrics.dropwizard.io/3.2.2/, [Online; accessed 28-June-2017]
8. Curry, E.: Message-oriented middleware. Middleware for communications pp. 1–28 (2004)
9. Dan Deeth, Sandvine: Hbo goes down (2014), http://www.internetphenomena.com/2014/03/hbo-goes-down/, [Online; accessed 15-March-2017]
10. Gilly, K., Juiz, C., Puigjaner, R.: An up-to-date survey in web load balancing. World Wide Web 14(2), 105–131 (2011)
11. Gilmore, S., Hillston, J., Ribaudo, M.: An efficient algorithm for aggregating pepa models. Ieee Transactions On Software Engineering 27(5), 449–464 (2001)
12. Gilmore, S., Hillston, J.: The pepa workbench: A tool to support a process algebra-based approach to performance modelling. Computer performance evaluation modelling techniques and tools pp. 353–368 (1994)
13. 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)
14. 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)
15. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
16. Lewis, J., Fowler, M.: Microservices (2014), martinfowler.com, [Online; accessed 5-March-2017]
17. Mell, P., Grance, T.: The nist definition of cloud computing (2011)
18. Microsoft: Service bus documentation (2017), https://docs.microsoft.com/en-us/azure/service-bus/, [Online; accessed 6-March-2017]
19. Microsoft: Virtual machine scale sets (2017), https://azure.microsoft.com/en-us/services/virtual-machine-scale-sets/, [Online; accessed 5-March-2017]
20. 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]
21. Pivotal: Spring (2017), http://spring.io/, [Online; accessed 28-June-2017]
22. 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]
23. 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]
24. Shephard, S.: Performance modelling and simulation of skewed demand in complex systems (2017), https://github.com/sshephard2/skewed-modelling, [Online; accessed 28-June-2017]
25. 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]
26. Thomas, N., Hillston, J.: Using Markovian process algebra to specify interactions in queueing systems. University of Edinburgh (1997)