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, load-balancing, 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 our system will help ensure our 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.

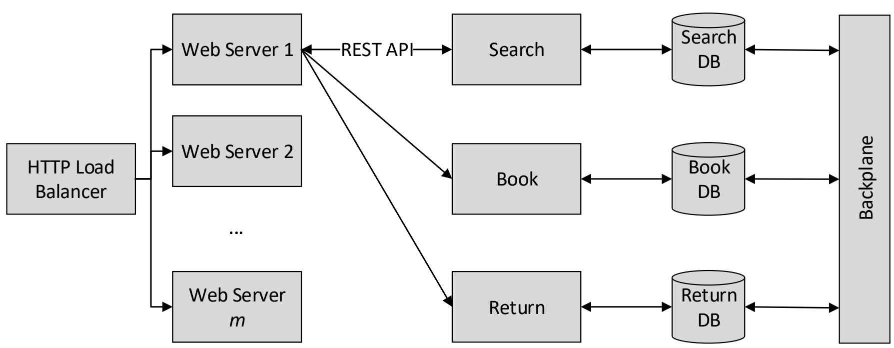
Point-to-Point Queues. Azure Queues are a point-to-point service implementing First In, First Out (FIFO) message delivery. Many processes may send messages to a queue, and each message is received by one consumer - though it may be one of several consumers competing for messages from this queue. This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

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 our use case requirements - search for tickets, book tickets, return tickets - might 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. We would need a lower level of granularity however to deal with skewed demand for a particular type of ticket, perhaps a separate microservice for booking each 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].

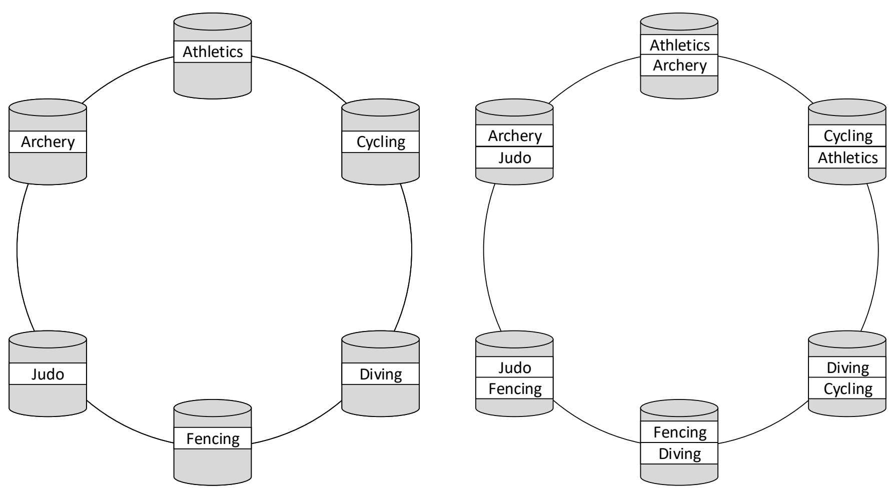
In our use case, the quantity of data does not approach the levels of "Big Data" applications. We are interested in partitioning as a means of scaling the demand for that data. Our ticketing system will not require a large number of columns and the three operations outlined do not have significantly different column requirements. 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.

The scalability of distributed databases usually comes at the price of a relaxed consistency model - so-called BASE (Basically Available, Soft state, Eventually Consistent) rather than ACID (Atomic, Consistent, Isolated, Durable) transactions. In our ticketing system, eventual consistency is clearly sufficient for the return ticket scenario - returned tickets do not have to be made immediately available for booking. Individual ticket bookings must exist on only one partition to prevent the same ticket being booked more than once. Eventual consistency between search and book operations requires the customer to tolerate the concept of “reservation” of a ticket for a short period until a booking can be confirmed [23][6].

Another issue to be aware of is *replication*. Most distributed databases offer replication of data from one partition to another for availability. In our 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.

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.

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.

Fig. 3. PEPA queue model

$$
\begin{aligned}
&\mathit{Website} \stackrel{\mathit{def}}{=} (\mathit{request}, r).\mathit{Website}\\
&\mathit{Worker} \stackrel{\mathit{def}}{=} (\mathit{service}, s).\mathit{Worker}\\
&\mathit{Queue_0} \stackrel{\mathit{def}}{=} (\mathit{request}, r).\mathit{Queue_1}\\
&\mathit{Queue_1} \stackrel{\mathit{def}}{=} (\mathit{service}, s).\mathit{Queue_0}\\
&\mathit{Website} \underset{\mathit{request}}{\bowtie} \mathit{Queue_0}[N] \underset{\mathit{service}}{\bowtie} \mathit{Worker}
\end{aligned}
$$

5 PEPA Component Models

5.1 Distributed database without replication

The PEPA model for a distributed database is shown in Figure 4.

Fig. 4. Distributed database PEPA model

$$\begin{aligned}
a &= 1.0 - 10.0\\
c &= 1.0\\
db &= 5.0\\
Website &\stackrel{\mathit{def}}{=} (book_a, a).Website + (book_c, c).Website\\
DB_1 &\stackrel{\mathit{def}}{=} (book_a, \top).DBsrv_1\\
DBsrv_1 &\stackrel{\mathit{def}}{=} (dbsrv_1, \top).DB_1\\
DB_2 &\stackrel{\mathit{def}}{=} (book_c, \top).DBsrv_2\\
DBsrv_2 &\stackrel{\mathit{def}}{=} (dbsrv_2, \top).DB_2\\
Service_1 &\stackrel{\mathit{def}}{=} (dbsrv_1, db).Service_1\\
Service_2 &\stackrel{\mathit{def}}{=} (dbsrv_2, db).Service_2\\
Website &\underset{book_a, book_c}{\bowtie} DB_1 \parallel DB_2 \underset{dbsrv_1, dbsrv_2}{\bowtie} Service_1 \parallel Service_2
\end{aligned}$$

See the experimental results in Table 1.

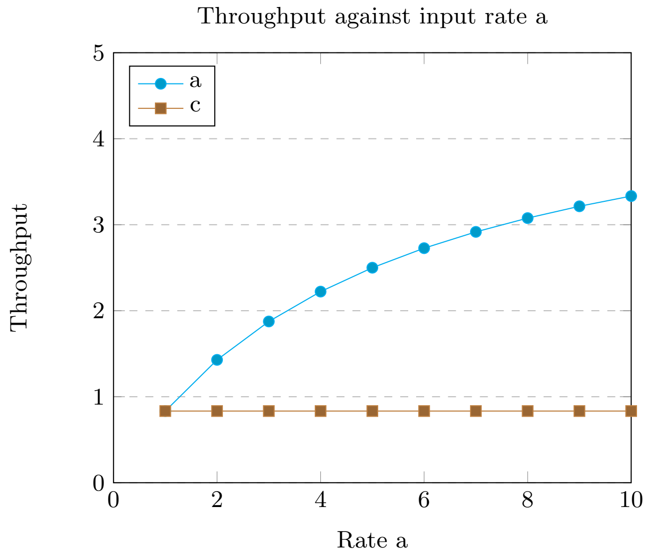
Table 1. Distributed database experimental results

Rate	Throughput			
a	$\mathrm{book_a}$	$\mathrm{book_c}$	$\mathrm{dbsrv_1}$	$\mathrm{dbsrv_2}$
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

5.2 Distributed database with replication

See the experimental results in Table 2.

Fig. 5. Distributed database experimental results



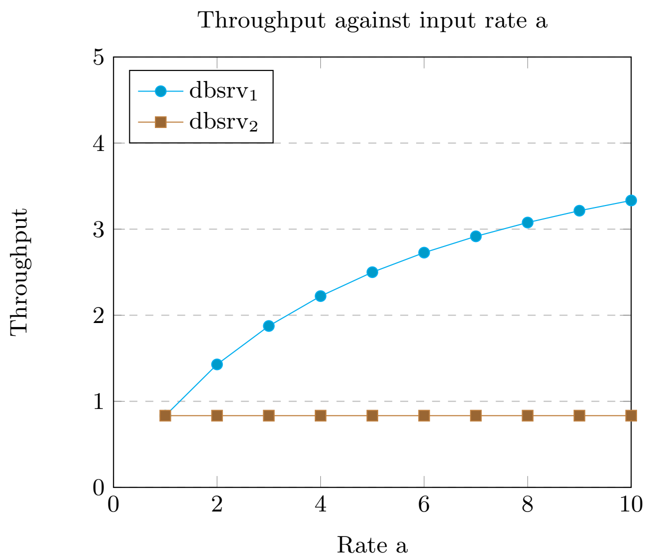


Fig. 6. Distributed database with replication PEPA model

$$
\begin{aligned}
a &= 1.0 - 10.0\\
c &= 1.0\\
d &= 1.0\\
db &= 5.0\\
\mathit{Website} &\stackrel{\mathit{def}}{=} (\mathit{book}_a, a).\mathit{Website} + (\mathit{book}_c, c).\mathit{Website} + (\mathit{book}_d, d).\mathit{Website}\\
\mathit{DB_1} &\stackrel{\mathit{def}}{=} (\mathit{book}_a, \top).\mathit{DBsrv_1} + (\mathit{book}_c, \top).\mathit{DBsrv_1}\\
\mathit{DBsrv_1} &\stackrel{\mathit{def}}{=} (\mathit{dbsrv_1}, \top).\mathit{DB_1}\\
\mathit{DB_2} &\stackrel{\mathit{def}}{=} (\mathit{book}_c, \top).\mathit{DBsrv_2} + (\mathit{book}_d, \top).\mathit{DBsrv_2}\\
\mathit{DBsrv_2} &\stackrel{\mathit{def}}{=} (\mathit{dbsrv_2}, \top).\mathit{DB_2}\\
\mathit{DB_3} &\stackrel{\mathit{def}}{=} (\mathit{book}_d, \top).\mathit{DBsrv_3} + (\mathit{book}_a, \top).\mathit{DBsrv_3}\\
\mathit{DBsrv_3} &\stackrel{\mathit{def}}{=} (\mathit{dbsrv_3}, \top).\mathit{DB_3}\\
\mathit{Service_1} &\stackrel{\mathit{def}}{=} (\mathit{dbsrv_1}, db).\mathit{Service_1}\\
\mathit{Service_2} &\stackrel{\mathit{def}}{=} (\mathit{dbsrv_2}, db).\mathit{Service_2}\\
\mathit{Service_3} &\stackrel{\mathit{def}}{=} (\mathit{dbsrv_3}, db).\mathit{Service_3}\\
\mathit{Website} &\underset{book_a, book_c, book_d}{\bowtie} \mathit{DB_1} \parallel \mathit{DB_2} \parallel \mathit{DB_3} \underset{dbsrv_1, dbsrv_2, dbsrv_3}{\bowtie} \mathit{Service_1} \parallel \mathit{Service_2} \parallel \mathit{Service_3}
\end{aligned}
$$

Table 2. Distributed database with replication experimental results

Rate	Throughput					
a	$\mathrm{book_a}$	$\mathrm{book_c}$	$\mathrm{book_d}$	$\mathrm{dbsrv_1}$	$\mathrm{dbsrv_2}$	$\mathrm{dbsrv_3}$
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

5.3 Shared middleware queue

Queues have already been extensively modelled in PEPA [26]...

Fig. 7. Distributed database with replication experimental results

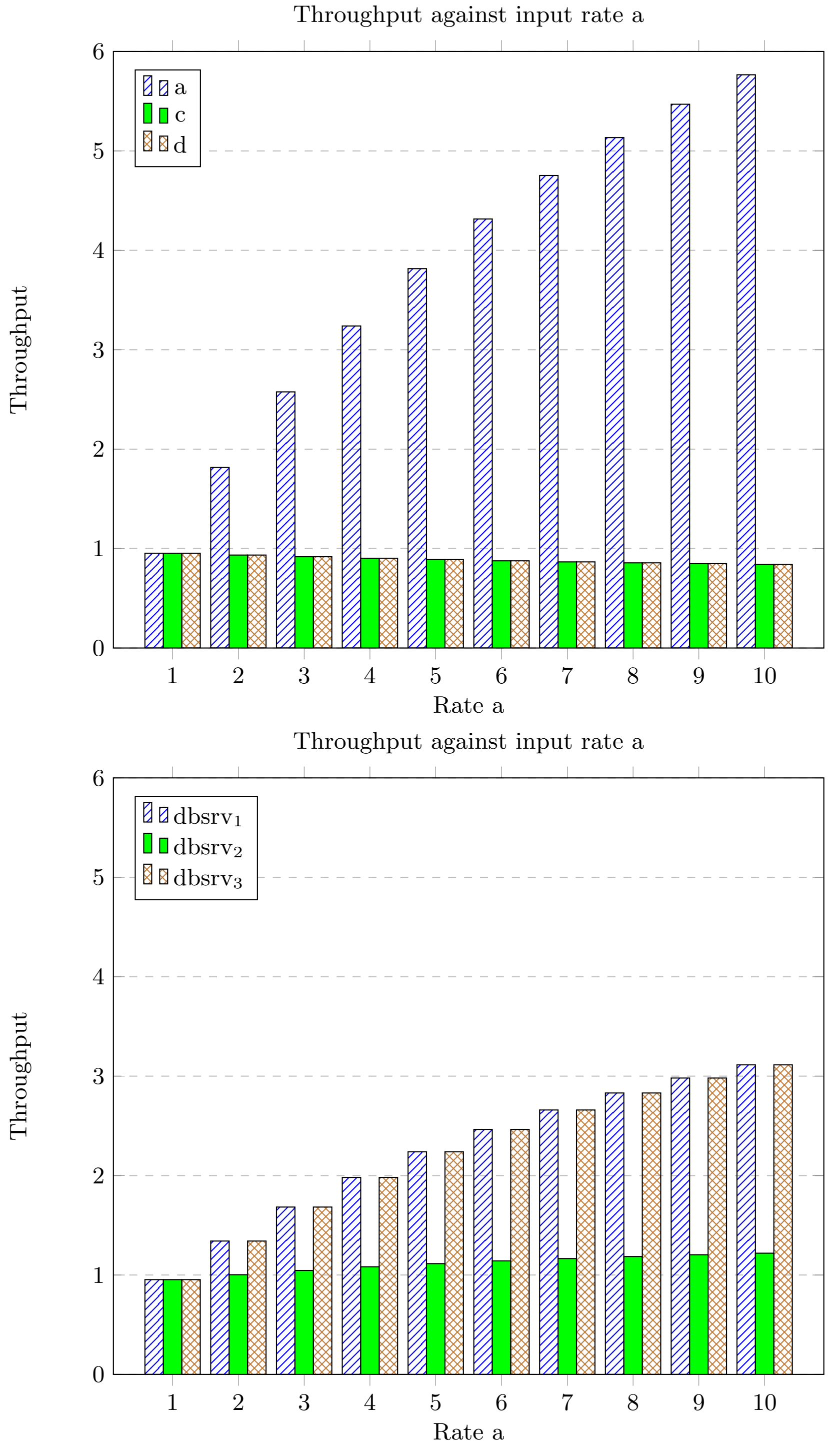


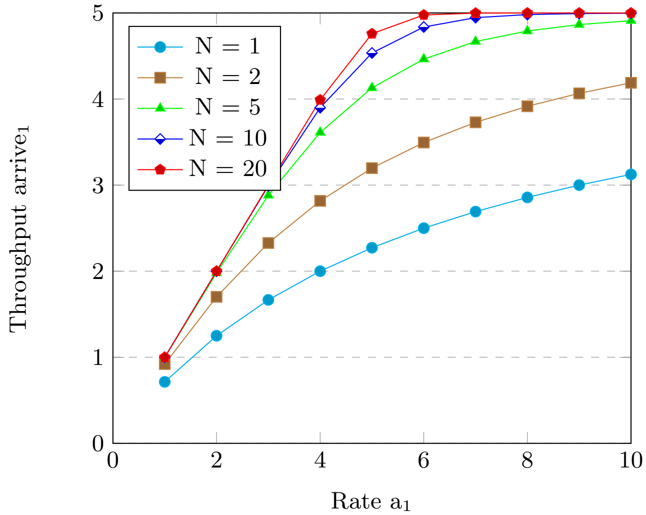
Fig. 8. Generic shared queue PEPA model

$$
\begin{aligned}
a_1 &= 1.0 - 10.0\\
s_1 &= 5.0\\
a_2 &= 1.0\\
s_2 &= 5.0\\
Arrival_1 &\stackrel{def}{=} (arrive_1, a_1).Arrival_1\\
Service_1 &\stackrel{def}{=} (serve_1, s_1).Service_1\\
Arrival_2 &\stackrel{def}{=} (arrive_2, a_2).Arrival_2\\
Service_2 &\stackrel{def}{=} (serve_2, s_2).Service_2\\
Q_0 &\stackrel{def}{=} (arrive_1, \top).Q_1 + (arrive_2, \top).Q_2\\
Q_1 &\stackrel{def}{=} (serve_1, \top).Q_0\\
Q_2 &\stackrel{def}{=} (serve_2, \top).Q_0
\end{aligned}
$$

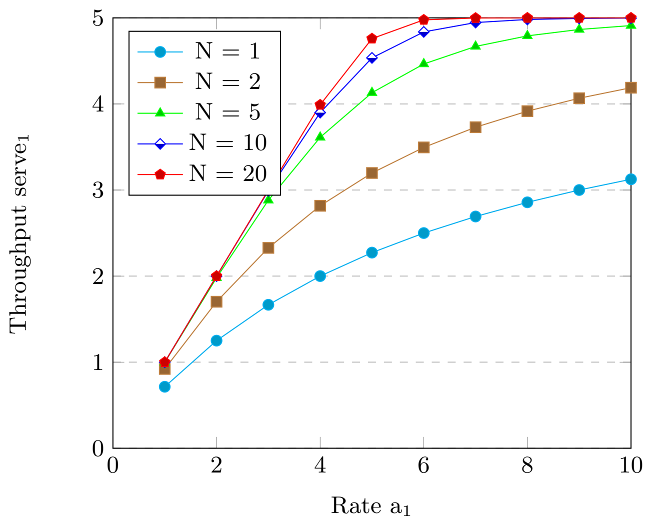
$$Arrival_1 \underset{arrive_1}{\bowtie} Q_0[N] \underset{serve_1}{\bowtie} Service_1 \underset{arrive_2}{\bowtie} Arrival_2 \underset{serve_2}{\bowtie} Service_2$$

Fig. 9. Generic shared queue experimental results

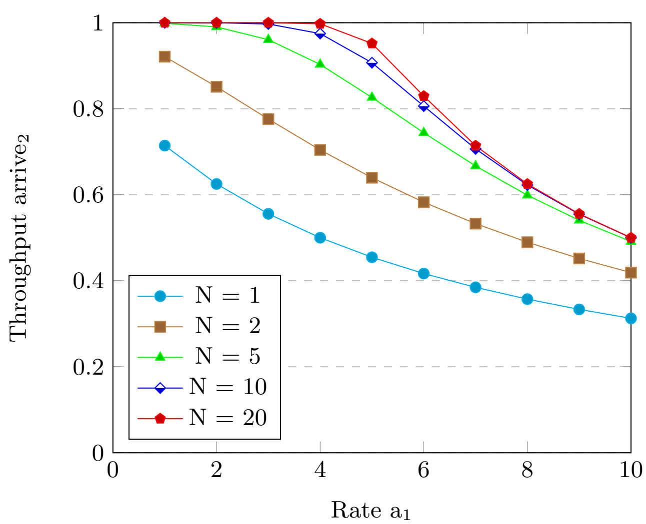
Throughput of $arrive_1$ against input rate a_1 for different queue lengths N



Throughput of $serve_1$ against input rate a_1 for different queue lengths N



Throughput of $arrive_2$ against input rate a_1 for different queue lengths N



6 System architectures

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.

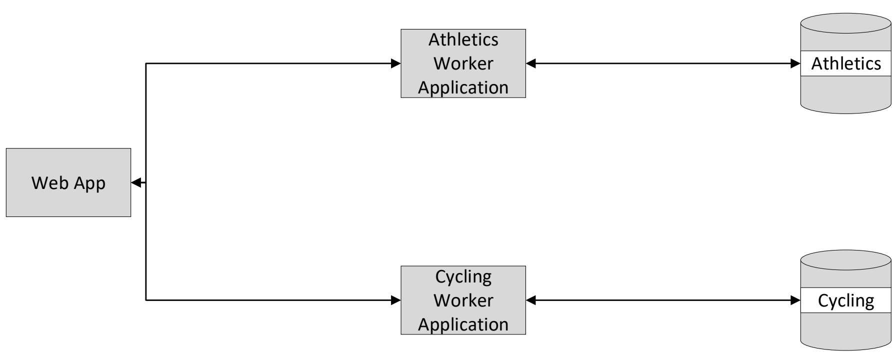
For each architecture it will be assumed that the web application will be designed to cope with the required demand, using a cluster of web servers where the throughput is managed using some HTTP Load Balancing algorithm [10], and potentially Elastic Scaling of servers e.g. using the autoscaling features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

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.

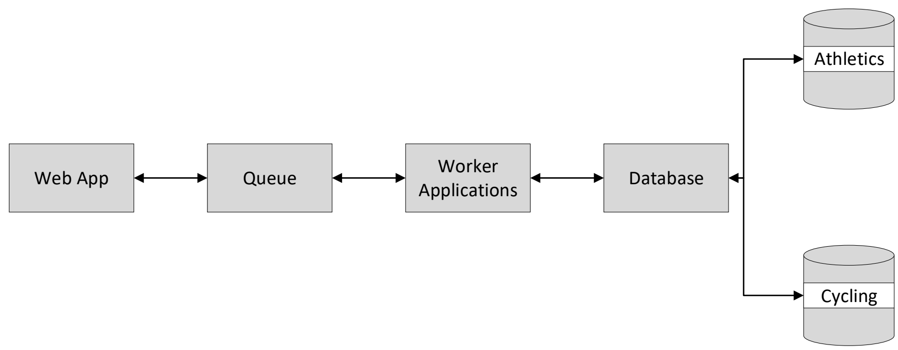
Fig. 10. Simple microservices architecture



6.2 Shared queue middleware

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

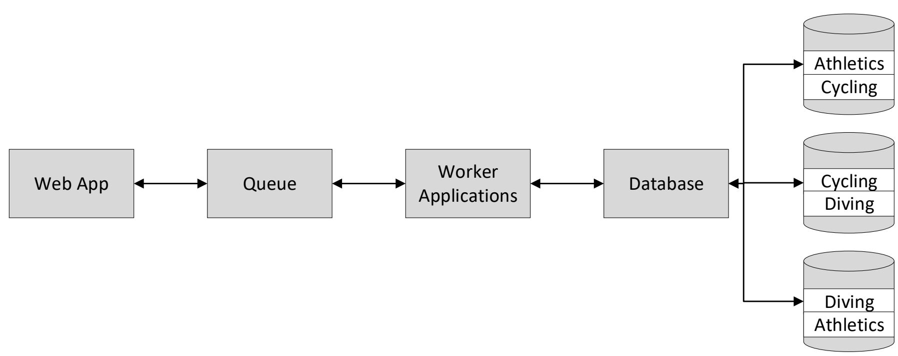
Fig. 11. Shared queue middleware architecture



6.3 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.

Fig. 12. Distributed database with replication architecture



7 PEPA System Models

7.1 Simple microservices

Fig. 13. Simple microservices PEPA model

$$
\begin{aligned}
a &= 1.0 - 10.0\\
c &= 1.0\\
w &= 100.0\\
db &= 6.5\\
\mathit{Website} &\stackrel{\mathit{def}}{=} (\mathit{athletics}, a).\mathit{Website} + (\mathit{cycling}, c).\mathit{Website}\\
\mathit{Worker}_A &\stackrel{\mathit{def}}{=} (\mathit{athletics}, \top).\mathit{WorkerSrv}_A\\
\mathit{WorkerSrv}_A &\stackrel{\mathit{def}}{=} (\mathit{workerA}, \top).\mathit{Worker}_A\\
\mathit{Worker}_C &\stackrel{\mathit{def}}{=} (\mathit{cycling}, \top).\mathit{WorkerSrv}_C\\
\mathit{WorkerSrv}_C &\stackrel{\mathit{def}}{=} (\mathit{workerC}, \top).\mathit{Worker}_C\\
\mathit{DB}_1 &\stackrel{\mathit{def}}{=} (\mathit{workerA}, w).\mathit{DBsrv}_1\\
\mathit{DBsrv}_1 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv1}, db).\mathit{DB}_1\\
\mathit{DB}_2 &\stackrel{\mathit{def}}{=} (\mathit{workerC}, w).\mathit{DBsrv}_2\\
\mathit{DBsrv}_2 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv2}, db).\mathit{DB}_2\\
\mathit{Service}_1 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv1}, db).\mathit{Service}_1\\
\mathit{Service}_2 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv2}, db).\mathit{Service}_2
\end{aligned}
$$

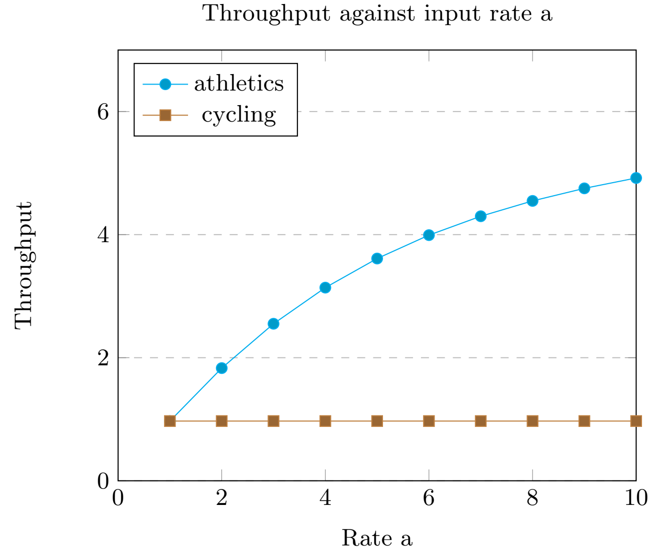
$$\mathit{Service}_1 \underset{\mathit{dbsrv1}}{\bowtie} \mathit{DB}_1 \underset{\mathit{workerA}}{\bowtie} \mathit{Worker}_A \underset{\mathit{athletics}}{\bowtie} \mathit{Website} \underset{\mathit{cycling}}{\bowtie} \mathit{Worker}_C \underset{\mathit{workerC}}{\bowtie} \mathit{DB}_2 \underset{\mathit{dbsrv2}}{\bowtie} \mathit{Service}_2$$

See the experimental results in Table 3.

Table 3. Simple microservices experimental results

Rate	Throughput					
a	athletics	cycling	dbsrv1	dbsrv2	workerA	workerC
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. 14. Simple microservices experimental results



7.2 Shared queue and distributed database

See the experimental results in Table 4.

Table 4. Shared queue and distributed database experimental results

Rate	Throughput					
a	$book_a$	$book_c$	$dbsrv_1$	$dbsrv_2$	$queue_a$	$queue_c$
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

7.3 Shared queue and distributed database with replication

See the experimental results in Table 5.

Fig. 15. Shared queue and distributed database

$$
\begin{aligned}
a &= 1.0 - 10.0 \\
c &= 1.0 \\
q &= 100.0 \\
db &= 5.0 \\
\mathit{Website} &\stackrel{\mathit{def}}{=} (\mathit{book}_a, a).\mathit{Website} + (\mathit{book}_c, c).\mathit{Website} \\
Q_0 &\stackrel{\mathit{def}}{=} (\mathit{book}_a, \top).Q_A + (\mathit{book}_c, \top).Q_C \\
Q_A &\stackrel{\mathit{def}}{=} (\mathit{queue}_a, \top).Q_0 \\
Q_C &\stackrel{\mathit{def}}{=} (\mathit{queue}_c, \top).Q_0 \\
\mathit{DB}_1 &\stackrel{\mathit{def}}{=} (\mathit{queue}_a, q).\mathit{DBsrv}_1 \\
\mathit{DBsrv}_1 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_1, db).\mathit{DB}_1 \\
\mathit{DB}_2 &\stackrel{\mathit{def}}{=} (\mathit{queue}_c, q).\mathit{DBsrv}_2 \\
\mathit{DBsrv}_2 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_2, db).\mathit{DB}_2 \\
\mathit{Service}_1 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_1, db).\mathit{Service}_1 \\
\mathit{Service}_2 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_2, db).\mathit{Service}_2 \\
\mathit{Website} &\underset{\mathit{book}_a,\mathit{book}_c}{\bowtie} Q_0[10.0] \underset{\mathit{queue}_a,\mathit{queue}_c}{\bowtie} \mathit{DB}_1 \parallel \mathit{DB}_2 \underset{\mathit{dbsrv}_1,\mathit{dbsrv}_2}{\bowtie} \mathit{Service}_1 \parallel \mathit{Service}_2
\end{aligned}
$$

Fig. 16. Shared queue and distributed database with replication

$$
\begin{aligned}
a &= 1.0 - 10.0 \\
c &= 1.0 \\
d &= 1.0 \\
q &= 100.0 \\
db &= 5.0 \\
\mathit{Website} &\stackrel{\mathit{def}}{=} (\mathit{book}_a, a).\mathit{Website} + (\mathit{book}_c, c).\mathit{Website} + (\mathit{book}_d, d).\mathit{Website} \\
Q_0 &\stackrel{\mathit{def}}{=} (\mathit{book}_a, \top).Q_A + (\mathit{book}_c, \top).Q_C + (\mathit{book}_d, \top).Q_D \\
Q_A &\stackrel{\mathit{def}}{=} (\mathit{queue}_a, \top).Q_0 \\
Q_C &\stackrel{\mathit{def}}{=} (\mathit{queue}_c, \top).Q_0 \\
Q_D &\stackrel{\mathit{def}}{=} (\mathit{queue}_d, \top).Q_0 \\
\mathit{DB}_1 &\stackrel{\mathit{def}}{=} (\mathit{queue}_a, q).\mathit{DBsrv}_1 + (\mathit{queue}_c, q).\mathit{DBsrv}_1 \\
\mathit{DBsrv}_1 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_1, \top).\mathit{DB}_1 \\
\mathit{DB}_2 &\stackrel{\mathit{def}}{=} (\mathit{queue}_c, q).\mathit{DBsrv}_2 + (\mathit{queue}_d, q).\mathit{DBsrv}_2 \\
\mathit{DBsrv}_2 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_2, \top).\mathit{DB}_2 \\
\mathit{DB}_3 &\stackrel{\mathit{def}}{=} (\mathit{queue}_d, q).\mathit{DBsrv}_3 + (\mathit{queue}_a, q).\mathit{DBsrv}_3 \\
\mathit{DBsrv}_3 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_3, \top).\mathit{DB}_3 \\
\mathit{Service}_1 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_1, db).\mathit{Service}_1 \\
\mathit{Service}_2 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_2, db).\mathit{Service}_2 \\
\mathit{Service}_3 &\stackrel{\mathit{def}}{=} (\mathit{dbsrv}_3, db).\mathit{Service}_3 \\
\mathit{Website} &\underset{\mathit{book}_a,\mathit{book}_c,\mathit{book}_d}{\bowtie} Q_0[10.0] \underset{\mathit{queue}_a,\mathit{queue}_c,\mathit{queue}_d}{\bowtie} \mathit{DB}_1 \parallel \mathit{DB}_2 \parallel \mathit{DB}_3 \underset{\mathit{dbsrv}_1,\mathit{dbsrv}_2,\mathit{dbsrv}_3}{\bowtie} \\
&\qquad \mathit{Service}_1 \parallel \mathit{Service}_2 \parallel \mathit{Service}_3
\end{aligned}
$$

Table 5. Shared queue and distributed database with replication experimental results

Rate	Throughput								
a	$book_a$	$book_c$	$book_d$	$dbsrv_1$	$dbsrv_2$	$dbsrv_3$	$queue_a$	$queue_c$	$queue_d$
1	1	1	1	1	1	1	1	1	1
2	2	1	1	1.46	1.08	1.46	2	1	1
3	3	1	1	1.92	1.16	1.92	3	1	1
4	4	1	1	2.38	1.24	2.38	4	1	1
5	5	1	1	2.84	1.32	2.84	5	1	1
6	5.98	1	1	3.29	1.4	3.29	5.98	1	1
7	6.93	0.99	0.99	3.72	1.47	3.72	6.93	0.99	0.99
8	7.76	0.97	0.97	4.09	1.51	4.09	7.76	0.97	0.97
9	8.4	0.93	0.93	4.38	1.51	4.38	8.4	0.93	0.93
10	8.83	0.88	0.88	4.56	1.47	4.56	8.83	0.88	0.88

7.4 Comparison

The system results are compared in Table 6.

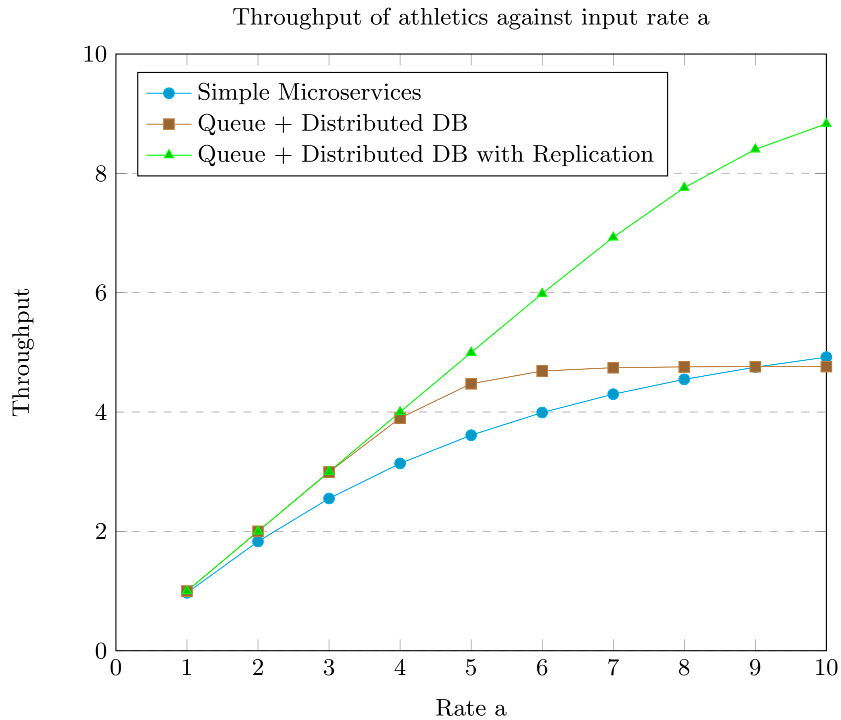
This shows that the simple microservices system does a good job of isolating the skewed demand from the rest of the system, but it is an inefficient use of the database resources. The actual throughput of the athletics demand is limited to its database's throughput, while the spare capacity of the cycling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

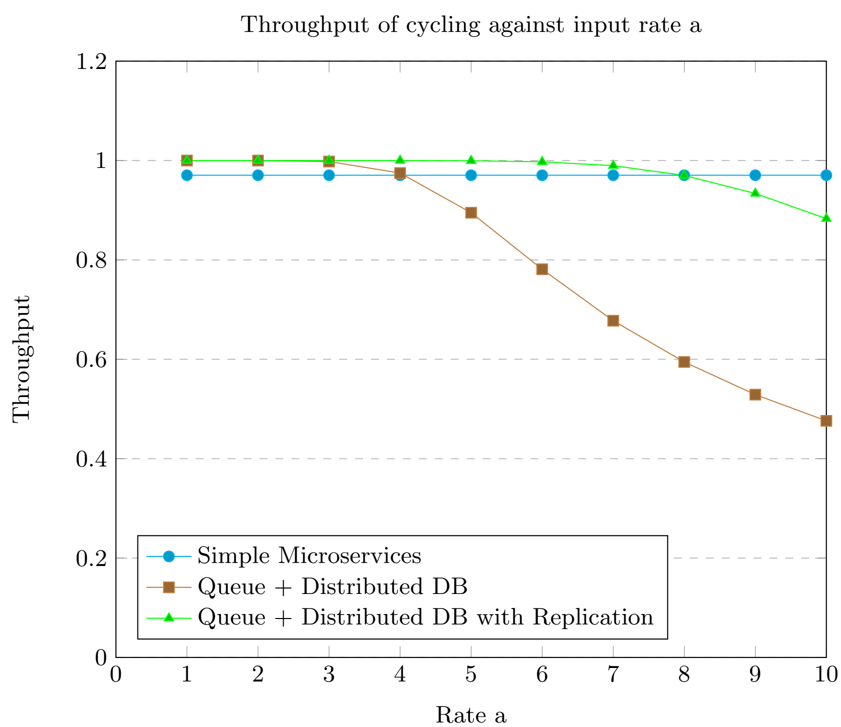
(NOTE: the replication model uses 3 nodes, the others use 2 - need to compare like with like. Try all with 3 or replication with 2?)

Table 6. 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. 17. Simple microservices experimental results





8 Built systems

Reference to github at [24]

General design decisions:

Cassandra [15][3] database. Create a Dbstress program and measured throughput of Cassandra node at 130 queries per second.

Measurement using Coda Hale Metrics [7].

Load testing using Apache JMeter [4].

8.1 Simple microservices

RESTful APIs using Java Spring [21].

Implemented a
control API which doesn't access the database.

Use JMeter with Poisson random timer (negative exponential distribution) with Cycling at a constant 10 threads/users and Athletics ramping up from 10-100 in steps of 10, so the desired demand is 20-200 requests per second.

Run the experiment 5 times and average the results (taking the maximum rolling 1 minute average for each number of users).

See the experimental results in Table 7.

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 7. 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

8.2 Shared queue middleware

See the experimental results in Table 8.

Fig. 18. Simple microservices experimental results

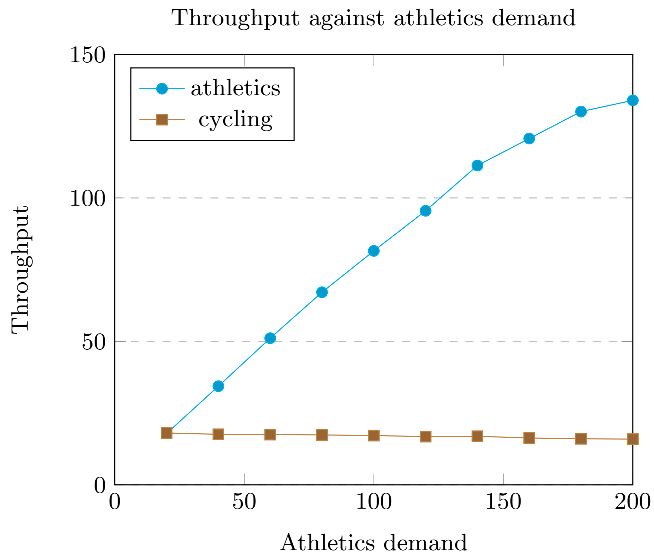


Fig. 19. Shared queue with distributed DB experimental results

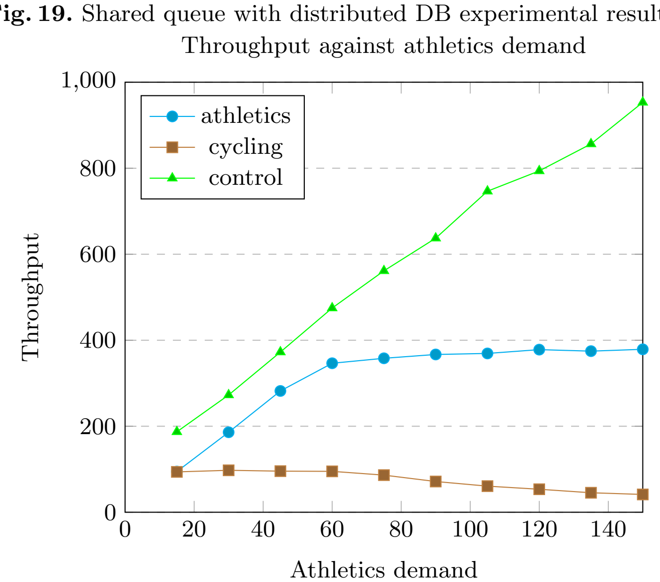


Table 8. Shared queue with distributed DB experimental results

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

8.3 Distributed database with replication

9 Conclusion and Future Work

...

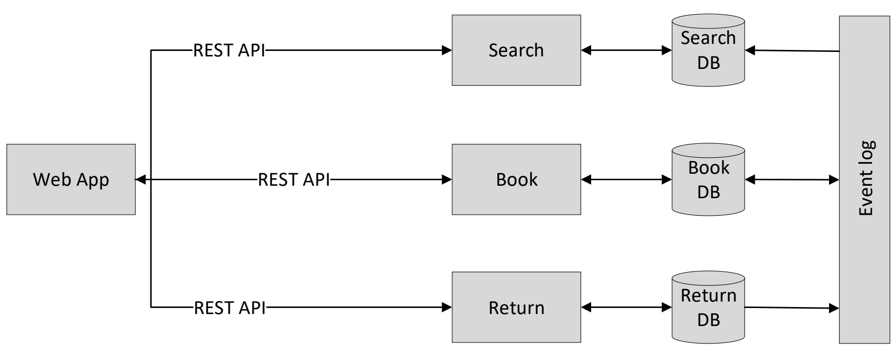
An interesting area of future work might be in using the modelling techniques in adaptive algorithms. A model might be used as a policy for elastic scaling, and compared with the performance of other right-sizing strategies; control theory, machine learning and other model based techniques including statistical.

9.1 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. 20. 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)