Investigating Cloud Technologies to Maximise Availability of Oversubscribed Resources

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. 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.

Keywords: Cloud, middleware, microservices, distributed databases, load-balancing, performance

1 Introduction

The London 2012 Olympic ticketing system was a high-profile example of an IT system unable to cope with customer demand. On initial launch, visitors to the site saw a "Sorry, we cannot process your request" message until demand had subsided [19]. The later launch of the ticket resale facility on the website suffered from outage of the process while new sales were still possible [13].

I propose that it is now possible to design and build a more resilient ticketing system through effective use of Cloud technologies where higher than normal demand for one function or ticket type would not block access to the others. The London 2012 Olympics have now passed into history, but such a ticketing system would have many applications today, and may be generalised to a system for allocating and releasing other resources with variable demand.

The problem of isolating the remainder of a system from parts that may be overloaded by demand also has relevance for Multi-tenanting, in Software as a Service products.

2 Background

I will consider an example system based on the Olympic ticketing use case above. Tickets will be for a multi-sport event, and each will consist of a ticket type (the sport), date, row, and seat number. The system will support three operations:

1. Search (for available tickets)
2. Book (allocate a ticket to a customer)

3. Return (customer releases a ticket allocation)

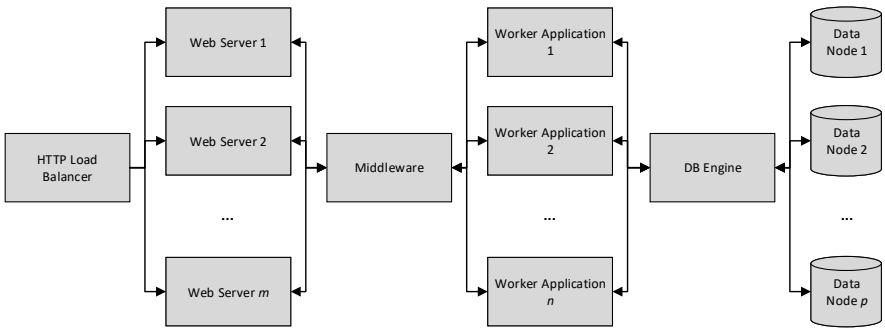
If it is not possible to search for or book tickets of one type because some component is overloaded due to demand, then the system should allow booking of other ticket types. It must also always be possible to return tickets of any type.

In this article, I am considering the problem of high demand for types of ticket. This is not about the number of tickets available and I will not consider issues of fair allocation of scarce resources. The demand may be:

1. predictable - we know which functions or ticket types are going to have the highest demand; or
2. unknown - we discover the areas of highest demand once the application goes online.

System Architecture. The proposed system will use a distributed architecture. 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.

Fig. 1. Ticketing application distributed architecture



3 Technology Review

We consider current cloud technologies that may be useful in distributing throughput generated by high demand throughout the system, and in decoupling its components from each other.

3.1 HTTP Load Balancing

HTTP load balancing improves the scalability of a web-based application by distributing the demand across multiple web servers. It also improves the availability of the overall system by directing HTTP requests to other servers if a web server fails. It therefore plays an essential role in a high-throughput system where an individual web server could not meet the demand.

For our ticketing use case, we will assume that we are developing a greenfield rather than legacy system, and that we will be deploying homogeneous web servers.

Content-blind load balancing. Content-blind load balancing is unaware of the information contained in the HTTP requests. It can be carried out wholly within layers 2 and 3 of the OSI reference model [8] (Data Link or Network) according to one of the following algorithms [11].

Round Robin. Assign connections sequentially on a simple round-robin basis. In a homogeneous environment this may be unweighted - each server receives equal connections over time. With heterogeneous web servers, a weighted version of the algorithm distributes the percentage of traffic proportionately to the different capacities of the servers.

Least Connection. This is a dynamic scheduling algorithm that assigns each incoming connection to the web servers with the least number of current connections. Again, this may be weighted or unweighted.

Least Loaded. Assign connections to the web server that has the highest spare capacity, determined by an agent on each server that updates the load balancer on its capacity and utilisation.

Random Server Selection. Assign connections to web servers according to a uniform random sequence.

Content-aware load balancing. Content-aware load balancing can act based on the information contained in the HTTP requests. Routing may therefore depend on URIs identifying different parts of a web site or RESTful services, or even certain query parameters or POST payloads. In our use case this gives us a means of *isolating* demand for certain services rather than balancing it across all web servers.

Furthermore, as HTTP/1.1 permits persistent connections, several HTTP requests from a client may use the same TCP connection. This reduces server and network overheads resulting in a lower response time.

Content-aware load balancing must take place at layer 7 of the OSI model (Application i.e. HTTP). In most content-aware architectures, the load balancer forwards the HTTP response from the targeted web server back to the client. This makes the load balancer a bottleneck in the system.

Evaluation. Response time and web server utilisation are useful metrics for comparing different load balancing techniques [3]. An effective HTTP load balancer will produce a low response time, representing a desirable user experience, and evenly distributed utilisation of each web server, minimising the cost to the system owner.

3.2 Elastic Scaling

Rapid elasticity is an essential characteristic of Cloud Computing by the NIST definition [16]. Computing resources, in our ticketing system web servers, worker applications and even data nodes, 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.

Amazon Web Services [2] and Microsoft Azure [18] both offer autoscaling products. They have similar capabilities and I will consider the Amazon Web Services offering in more detail.

AWS Auto Scaling. In Amazon Web Services Auto Scaling, the system designer defines a logical group of EC2 instances (e.g. Web servers) with a minimum and maximum number of servers, and a Scaling Plan. This grouping of instances is a means of sharing load. Load may be isolated between parts of the system by organising them into different autoscaling groups.

Manual Scaling and Scheduled Scaling are available but most relevant for our use case is Dynamic Scaling, based on demand. An alarm is raised when the value of a metric breaches a threshold for a defined period. A policy then instructs Auto Scaling whether to respond to an alarm with single adjustments per alarm or adjustments based on the size of the threshold breach.

Scaling based on metrics. Alarms based on metrics from Amazon CloudWatch may trigger auto scaling. Status, Disk, Network and CPU Utilisation metrics are currently available. These may be measured per EC2 instance or statistically aggregated over the Auto Scaling group. For example, a scaling policy may provision an additional Web server when the average Web server CPU Utilisation rises above 80% in a system with uniform load balancing.

Scaling based on Amazon SQS. Amazon SQS is a message queueing system (see *Middleware*). Amazon Cloudwatch may also collect metrics on queues, the number of messages sent, received and deleted, the age of the oldest message in the queue and also the length of the queue i.e. the number of messages awaiting retrieval. The queue length increases with demand and decreases as a system services that demand, so a scaling policy may provision an additional Worker Application when the length of the queue supplying it breaches a threshold.

Launch Configurations and Lifecycle Hooks. In a scenario where a hard maximum of servers is set, say limited by budget, it would be possible to pre-provision

all the EC2 instances that may be required and switch them on and off with demand. For genuinely elastic scaling, it must be possible to provision instances dynamically. A *Launch Configuration* in AWS is a template for creating a new EC2 instance of a required configuration.

This may be sufficient for auto-scaling basic Web servers; there are off the shelf Amazon Machine Images available for the major web servers. For the Worker Applications however, we need to install bespoke software for our ticketing system on each instance. This is achieved via *Lifecycle Hooks* that call custom actions on instance launch or shutdown. Of course, this means it takes time after provisioning an additional Worker Application instance before it is usable. We need to factor this into the metric threshold and specify a cooldown period before the alarm triggers again.

Evaluation. An effective elastic scaling strategy will minimise both underutilisation and overutilisation of compute resources. Underutilisation indicates over-provision of instances, which is inefficient in energy usage and uneconomic in cost. Overutilisation may cause poor response time and even outage leading to loss of revenue. [10]

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 [5]. 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].

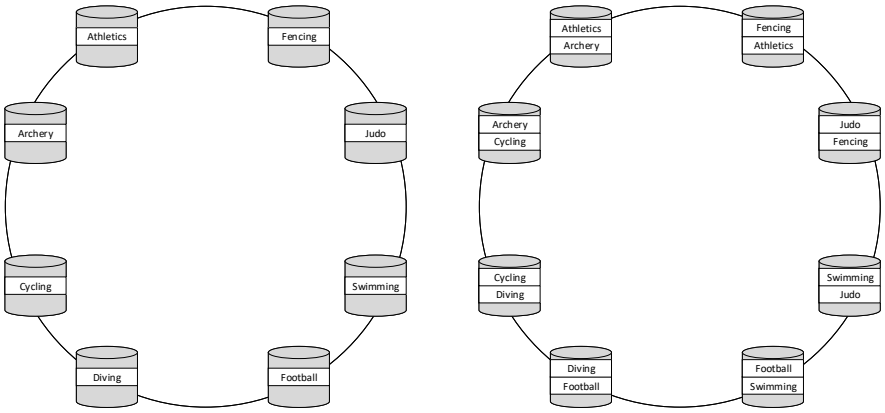
In our use case, the quantity of data does not approach the levels of "Big Data" applications. We are interested in partitioning as a means of scaling the demand for that data. Our ticketing system will not require a large number of columns and the three operations outlined do not have significantly 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. There may be an issue with the search operation, where tickets that appear as available at initial search turn out not to be available for booking. This is not ideal but not an uncommon user experience.

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



Evaluation. A successful partitioning strategy will ensure that an individual operation only uses a single data node; that every data node is equally used when demand is evenly distributed; and that any impact from skewed demand is limited to the directly affected data node.

3.4 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 Middleware. With synchronous middleware such as Remote Procedure Call (RPC) and middleware following RPC semantics, the calling process is blocked until the called service completes and returns control to the caller. These use object interfaces which result in tightly coupled components. This is undesirable for our ticketing application. However, they do guarantee sequential processing (i.e. that calls received are in the order they were sent, important for database transactions).

Asynchronous Middleware. 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[7].

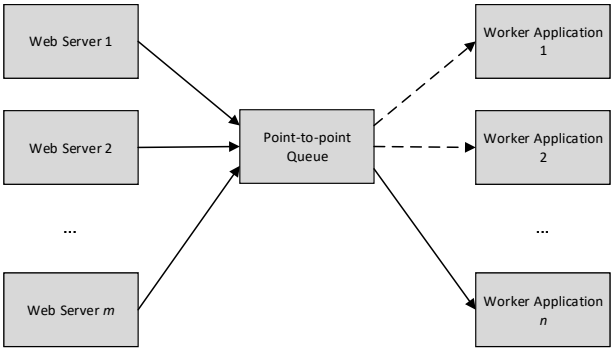
There are two main messaging models, both of which are offered by Microsoft Azure Service Bus [17] for example.

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.

Evaluation. The MOM services from Microsoft and Amazon provide tools for monitoring the number of messages in a queue or topic, and middleware can be examined by measuring whether a queue size reaches a steady state, or grows faster than downstream services can consume the messages. However, this is affected more by the services using the middleware than the choice of middleware itself. This is likely to be best achieved by end to end measurement.

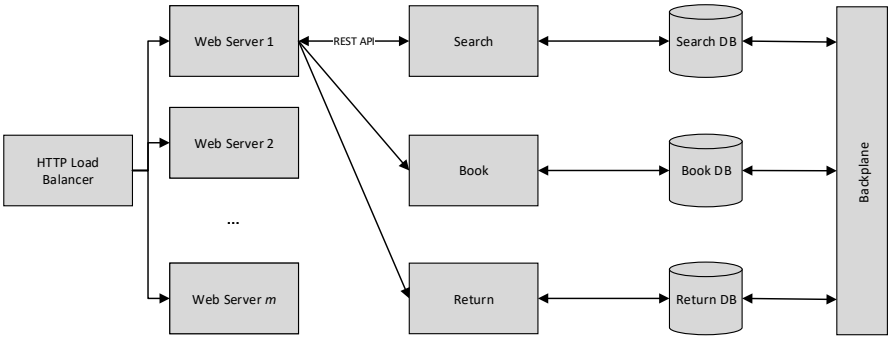
Fig. 3. Point-to-point queues



3.5 Microservices

Microservice architecture is an approach to structuring applications as suites of small services, defined by business capability verticals rather than technological layers [15] [20]. 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 [6]. 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. 4. Microservices



Evaluation. As for middleware, evaluation of different microservices approaches depends on end to end measurement. However, one area of interest is the efficiency of highly granular microservices. If the demand for the microservices is isolated then it is possible that some low demand services are underutilised.

4 Modelling

In our presentation of technologies relevant to the ticketing system use case, we have discussed criteria to evaluate each of them individually. In a distributed system we need to examine the end to end impacts. What are the optimum relative numbers of Web servers and Worker Applications for a given demand? Does removing a bottleneck at the Web or Worker Application layer shift the stress to the middleware, or the database? If we accept that some levels of demand cannot be met on a limited budget, and that some components will no longer meet the required throughput, how do we determine the impact on the remainder of the system?

Process Algebra. Process Algebras (such as PEPA or TIPP [12]) allow us to model throughput in interdependent processes, with a mixture of independent and shared actions operating at different rates. Each of our components can be described in this way, and queues have already been extensively modelled in PEPA [21]. The nature of process algebra as a mathematical language also means that it is possible to build a model of a whole system by composition of the component models.

Fig. 5. PEPA queue model

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

CloudSim. CloudSim [4] 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.

5 Conclusion and Future Work

In this article we showed that cloud technologies could manipulate the throughput at each of the layers of our ticketing system architecture.

Front-end. We can balance demand at the web front-end using content-blind HTTP load balancing, and isolate skewed demand using content-aware algorithms. Elastic scaling of web servers enables the front-end to respond to as much demand as the system owner is willing to pay for.

Application. We can decouple worker applications from the front-end using asynchronous middleware. Shared middleware balances the load, microservice architecture isolates it. The system can adapt to current demand by using elastic scaling to create or destroy worker applications, and by using scaling groups we can ensure that the number of each application type is appropriate to the demand.

Database. With care, we can use horizontal database partitioning to ensure that functions and/or data types are not shared between data nodes, isolating their demand from each other.

At the component level we can see whether an approach will balance or isolate load, or adapt to it, but at the system level we will need modelling techniques to predict the end to end throughput. We looked at two approaches, process algebra and programmatic, that could be used to build complex models from smaller components.

An interesting area of future work would be to create PEPA models for the components described here and to connect them together in two configurations of interest; balancing demand via shared middleware or isolating it via the microservices approach. We could build, instrument (e.g. using Byteman [9]) and measure a real system of each type to test the models, and then run them to see how each copes with different demand scenarios. For example, with rising skewed demand for one ticket type, at what point does the balanced demand approach begin to affect the entire system? Do either of the shared middleware and microservices approaches have clear efficiency advantages under certain conditions? A further area might be in using the modelling techniques as adaptive algorithms. Conceptually a CloudSim simulation might be used as a policy for elastic scaling.

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. Bryhni, H., Klovning, E., Kure, O.: A comparison of load balancing techniques for scalable web servers. IEEE network 14(4), 58–64 (2000)
4. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A., 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)

5. Cattell, R.: Scalable sql and nosql data stores. Acm Sigmod Record 39(4), 12–27 (2011)
6. Christian Posta, Red Hat: 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]
7. Curry, E.: Message-oriented middleware. Middleware for communications pp. 1–28 (2004)
8. Day, J.D., Zimmermann, H.: The osi reference model. Proceedings of the IEEE 71(12), 1334–1340 (1983)
9. Dinn, A.E.: Flexible, dynamic injection of structured advice using byteman. In: Proceedings of the tenth international conference on Aspect-oriented software development companion. pp. 41–50. Acm (2011)
10. Espadas, J., Molina, A., Jiménez, G., Molina, M., Ramírez, R., Concha, D.: A tenant-based resource allocation model for scaling software-as-a-service applications over cloud computing infrastructures. Future Generation Computer Systems 29(1), 273–286 (2013)
11. Gilly, K., Juiz, C., Puigjaner, R.: An up-to-date survey in web load balancing. World Wide Web 14(2), 105–131 (2011)
12. Götz, N., Herzog, U., Rettelbach, M.: Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In: Performance evaluation of computer and communication systems, pp. 121–146. Springer (1993)
13. James Pearce, BBC: London 2012: Olympics ticket resales website suspended (2012), http://www.bbc.co.uk/news/uk-16430850, [Online; accessed 2-March-2017]
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. Lewis, J., Fowler, M.: Microservices. martinfowler. com (2014)
16. Mell, P., Grance, T., et al.: The nist definition of cloud computing (2011)
17. Microsoft: Service bus documentation (2017), https://docs.microsoft.com/en-us/azure/service-bus/, [Online; accessed 6-March-2017]
18. Microsoft: Virtual machine scale sets (2017), https://azure.microsoft.com/en-us/services/virtual-machine-scale-sets/, [Online; accessed 5-March-2017]
19. 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]
20. 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]
21. Thomas, N., Hillston, J.: Using Markovian process algebra to specify interactions in queueing systems. University of Edinburgh (1997)