Using AWS Application Load Balancer and Network Load Balancer with EC2 Container Service

Nathan Peck
Oct 25, 2017 · 11 min read

Amazon Web Services recently released new second generation load balancers: Application Load Balancer (ALB), and Network Load Balancer (NLB). This was accompanied by a rename of the previous generation of load balancer to Classic Load Balancer. Understanding what these two new load balancers do and how they work will allow you to utilize the advantages of their new features, particularly their deep integration with EC2 Container Service to route traffic to your docker containers running in the cloud.

What is a load balancer?

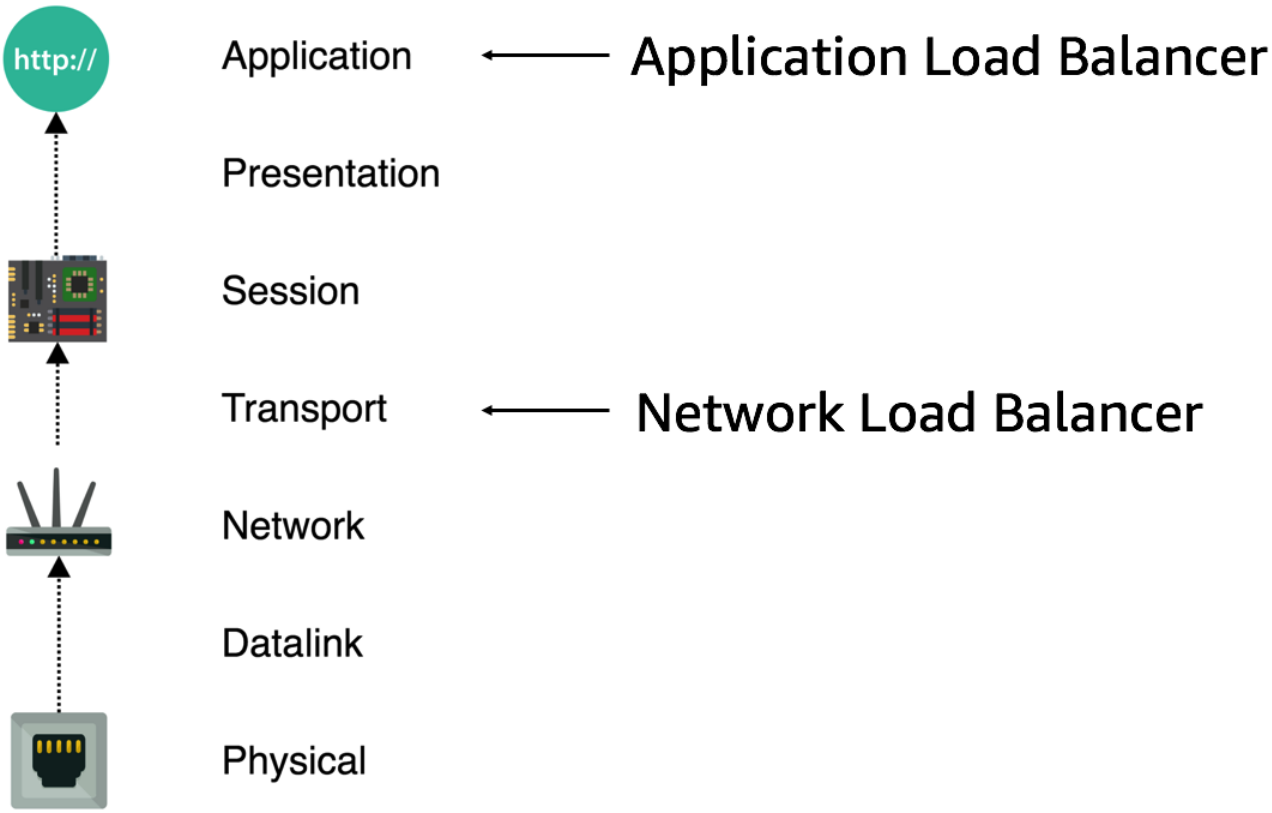
A load balancer accepts incoming network traffic from a client, and based on some criteria in the traffic it distributes those communications out to one or more backend servers:

Load balancers are key to building great internet applications, because they give your application the following benefits:

- **Redundancy** (One application server could die, but as long as there is at least one application server left the load balancer can still direct client traffic to the remaining working application server.)
- **Scalability** (By running two servers behind a load balancer you can now handle 2x the traffic from clients. Load balancers make it easy to add more and more backend servers as your traffic increases.)

What is the difference between an ALB and an NLB?

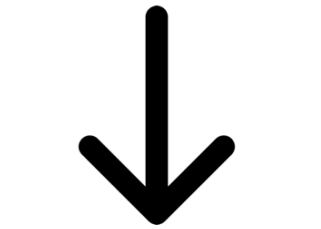
If you read the official AWS documentation for Application Load Balancer and Network Load Balancer you will notice that ALB is referred to as a "level 7" load balancer, while NLB is referred to as a "level 4" load balancer. These levels are a reference to the Open Systems Interconnection (OSI) model:

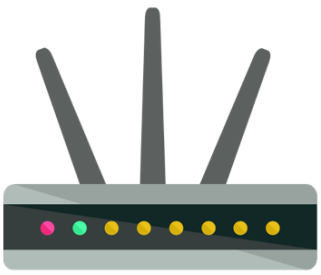


The OSI model categorizes the various operations that are involved in getting a network communication from one computer program on one machine to another computer program on another machine. Fully explaining the OSI model could fill an entire book, but the following diagram is a high level demonstration of the flow of network traffic down and back up the OSI model layers:

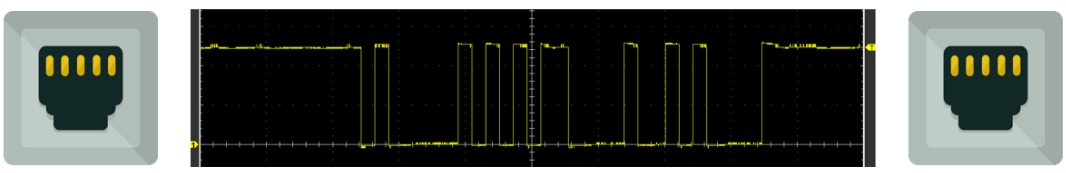

```
GET /hello-world.html HTTP/1.1
Host: www.test.com
Accept: text/html, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
(compatible; MSIE 6.0; Windows NT
5.1)
```

```
00000000  63 61 64 65 34 30 32 38 32 33 37 61 38 38 62 65  cade4028237a88be
00000010  35 34 63 30 61 39 61 38 65 31 39 31 62 36 33 63  54c0a9a8e191b63c
00000020  38 65 32 31 36 39 36 39 30 32 66 62 34 36 34 35  8e21696902fb4645
00000030  33 39 62 63 30 33 37 64 38 64 32 63 38 61 33 36  39bc037d8d2c8a36
00000040  61 61 34 64 63 66 38 39 63 35 62 32 66 64 64 38  aa4dcf89c5b2fdd8
00000050  62 36 64 35 64 32 61 39 31 38 65 66 30 36 62 30  b6d5d2a918ef06b0
00000060  31 35 66 38 31 32 64 61 38 33 33 39 63 61 38 63  15f812da8339ca8c
```

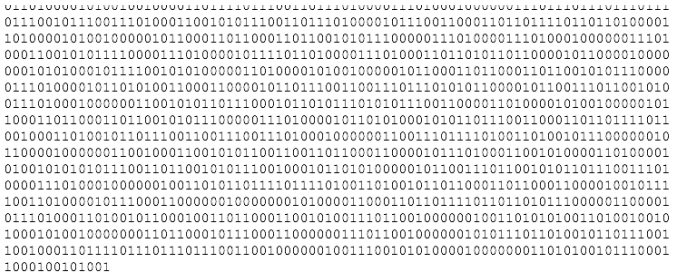


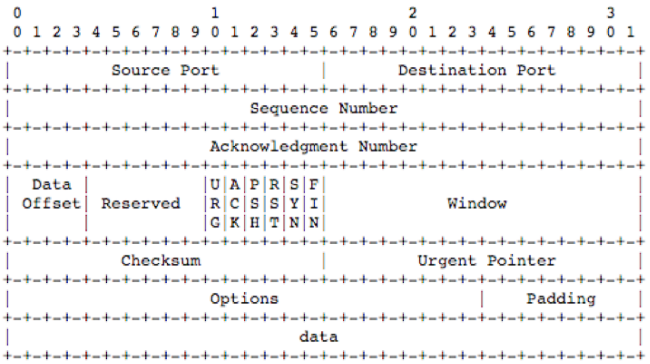


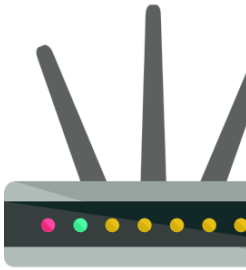
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

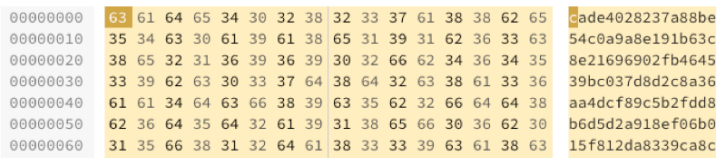


```
01000111010001010101010000100000001011110110100001100101011011000110110001101110
01011010111011101111011100100110110001100100001011100110100001110100011011010 1
10110000100000010010000101010001010100001010000001011110011000100101110001100010 00
01101000010100100100001101111011100110111010000011101000100000011101110111011011 1
```









00000000	63	61	64	65	34	30	32	38	32	33	37	61	38	38	62	65	cade4028237a88be
00000010	35	34	63	30	61	39	61	38	65	31	39	31	62	36	33	63	54c0a9a8e191b63c
00000020	38	65	32	31	36	39	36	39	30	32	66	62	34	36	34	35	8e21696902fb4645
00000030	33	39	62	63	30	33	37	64	38	64	32	63	38	61	33	36	39bc037d8d2c8a36
00000040	61	61	34	64	63	66	38	39	63	35	62	32	66	64	64	38	aa4dcf89c5b2fdd8
00000050	62	36	64	35	64	32	61	39	31	38	65	66	30	36	62	30	b6d5d2a918ef06b0
00000060	31	35	66	38	31	32	64	61	38	33	33	39	63	61	38	63	15f812da8339ca8c

```
GET /hello-world.html HTTP/1.1
Host: www.test.com
Accept: text/html, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
(compatible; MSIE 6.0; Windows NT
5.1)
```

1. At the application layer a web browser constructs an HTTP request, which is a small text document describing what resource the browser wants to fetch from a web server.

2. In order to make sure the communication between browser and server is secure the web request is encrypted using SSL/TLS. This process takes a server's public key and uses it to turn the HTTP payload into an unreadable chunk of encrypted binary data.

3. A few layers below at the transport layer the encrypted payload is split up into TCP packets. Each packet is a piece of the HTTP payload wrapped in metadata such as the source IP address that the packet originated from and the destination IP address where the packet should go.

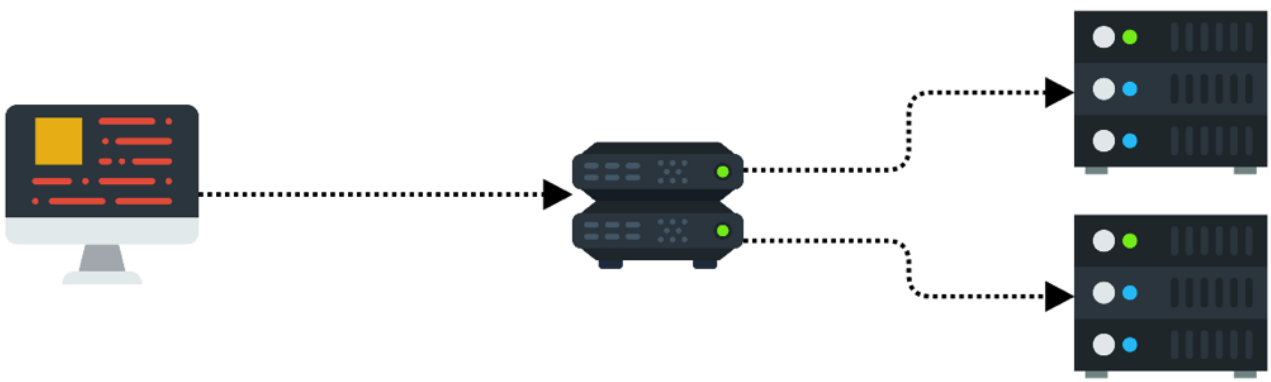
4. The physical layer takes the raw digital 1's and 0's that make up TCP packets and turns them into an analog signal such as an electrical pulse on a copper wire, a light pulse in a fiber optic cable, or a radio wave in the air. On the other end another device turns that analog signal back into a digital 1's and 0's.

5. The network traffic starts its trip back up the network stack as the 1's and 0's are interpreted into packets, which are then reassembled into the original encrypted data payload.

6. The server uses its private key to decrypt the SSL/TLS encrypted payload back into the original plaintext HTTP request document.

7. The server is able to interpret the plaintext HTTP request and figure out what resource to deliver back over the network.

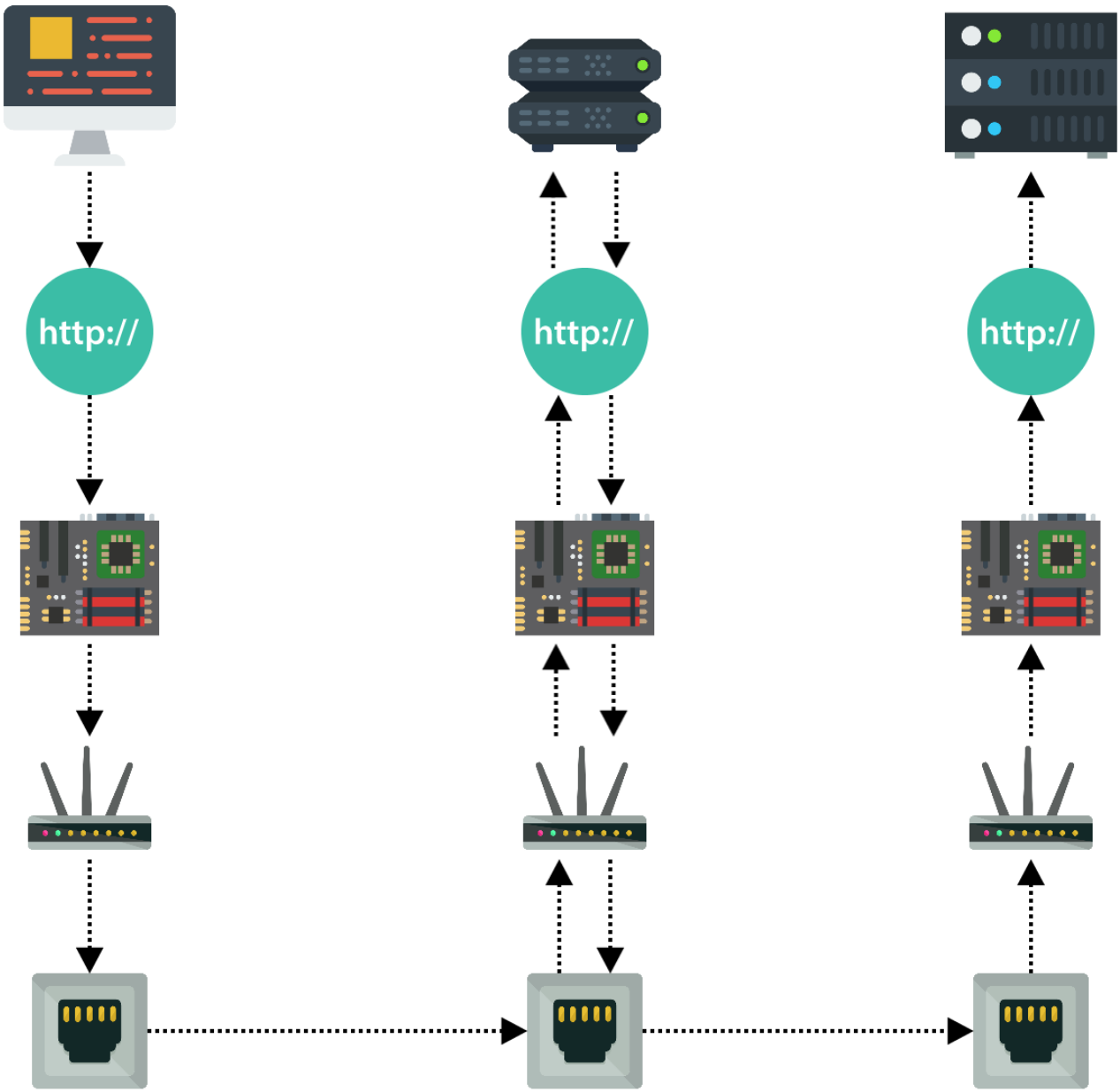
So with this understanding of how a network communication travels across the network it is clear that a load balancer is more than just this:



Those simple lines between client and load balancer and between load balancer and backend server are hiding a lot of the complexity of network communication. In reality the client has to send its communication down the stack, the communication travels back up the stack when it reaches the load balancer, and then the load balancer sends the communication back down the stack again to direct it to the destination server, where it finally travels back up the stack again.

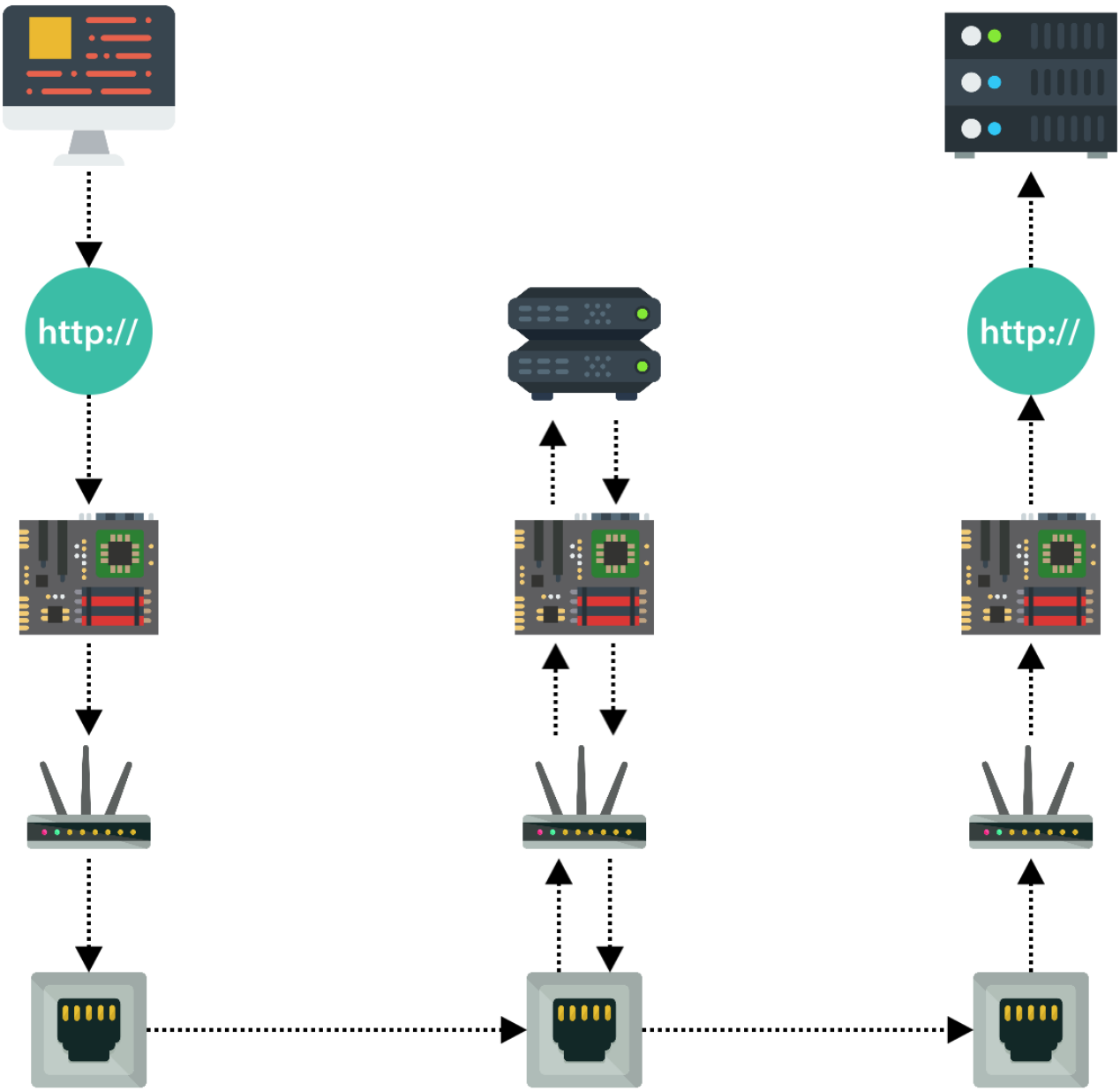
The “level” of a load balancer refers to how far back up the network stack a network communication must travel before the load balancer can direct it on its way back down the stack towards its final destination.

For a level 7 application load balancer network communication looks like this:



The network communication must travel all the way back up to the application layer, and the application load balancer reads the HTTP request to determine where it should be directed.

A level 4 network load balancer looks like this:



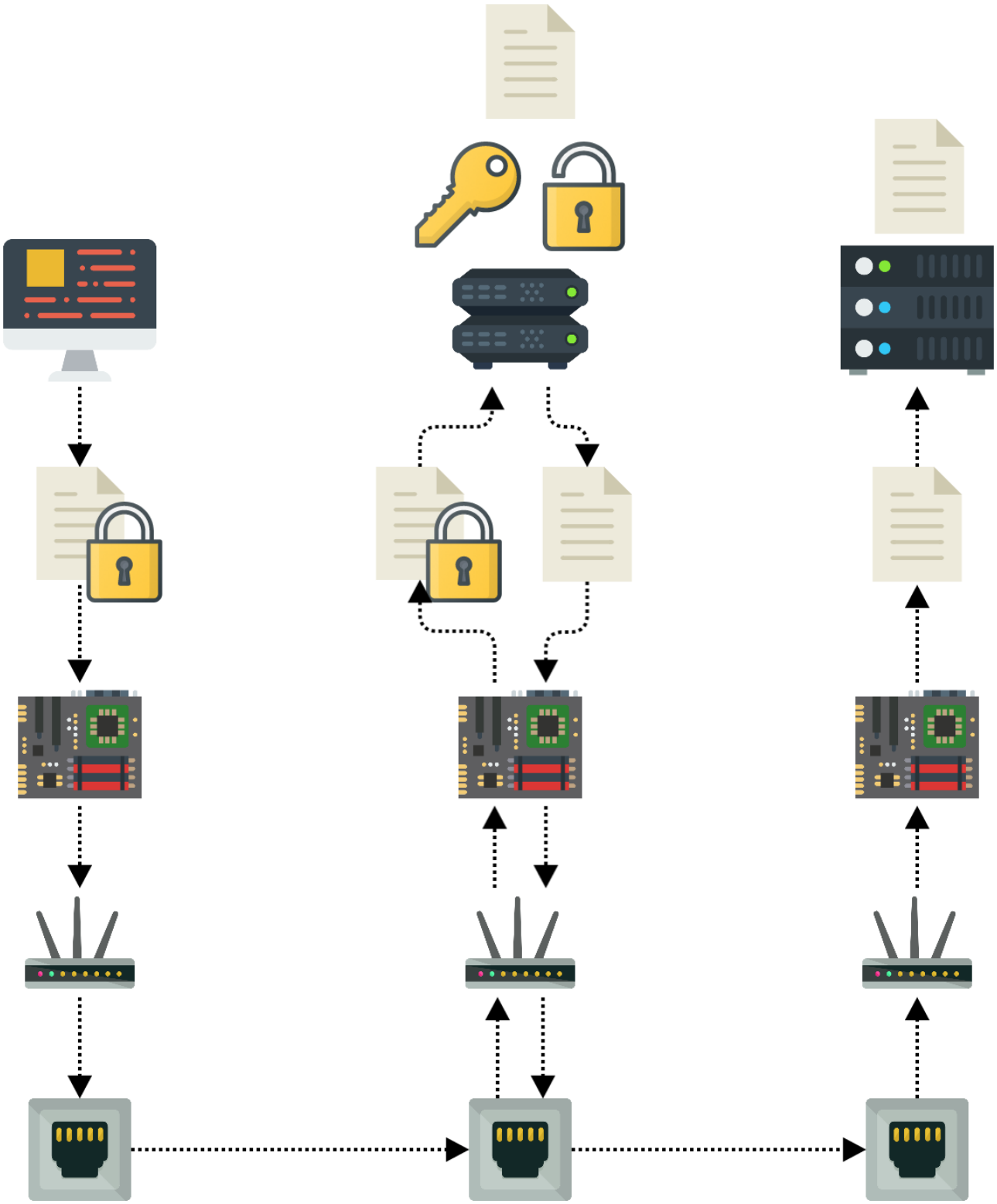
The network communication travels back up the stack to the transport layer, and the network load balancer reads the TCP packet information to direct the communication to the right place, but it does not actually read the information contained in those packets.

ALB and NLB Specific Features

The level of the load balancer that you use for your application controls what you can do with that load balancer, and make the different load balancers better suited for particular design patterns.

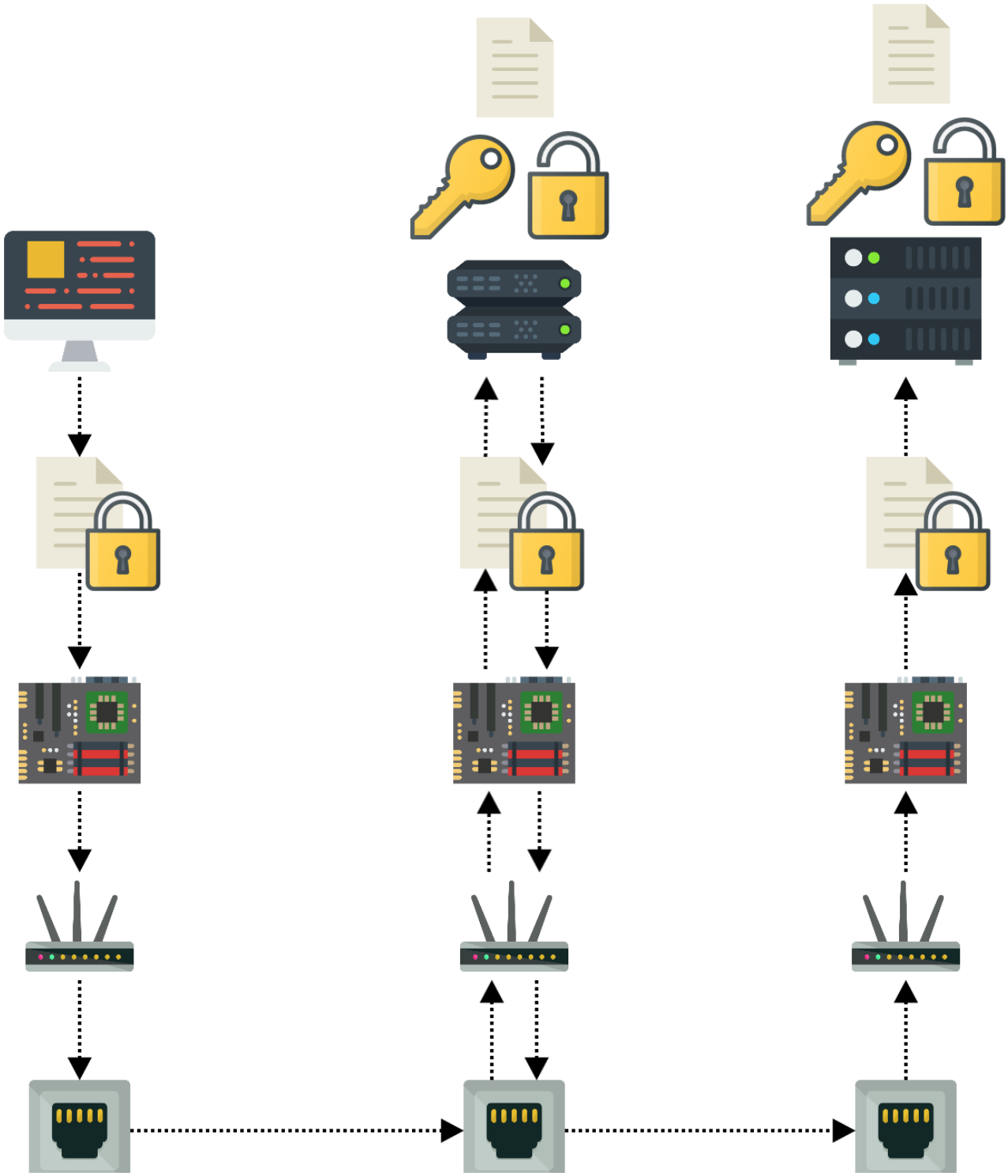
SSL/TLS

There are a few approaches to implementing SSL/TLS encryption for network traffic between a client and web server. For example, one common use for a level 7 application load balancer is SSL/TLS termination:



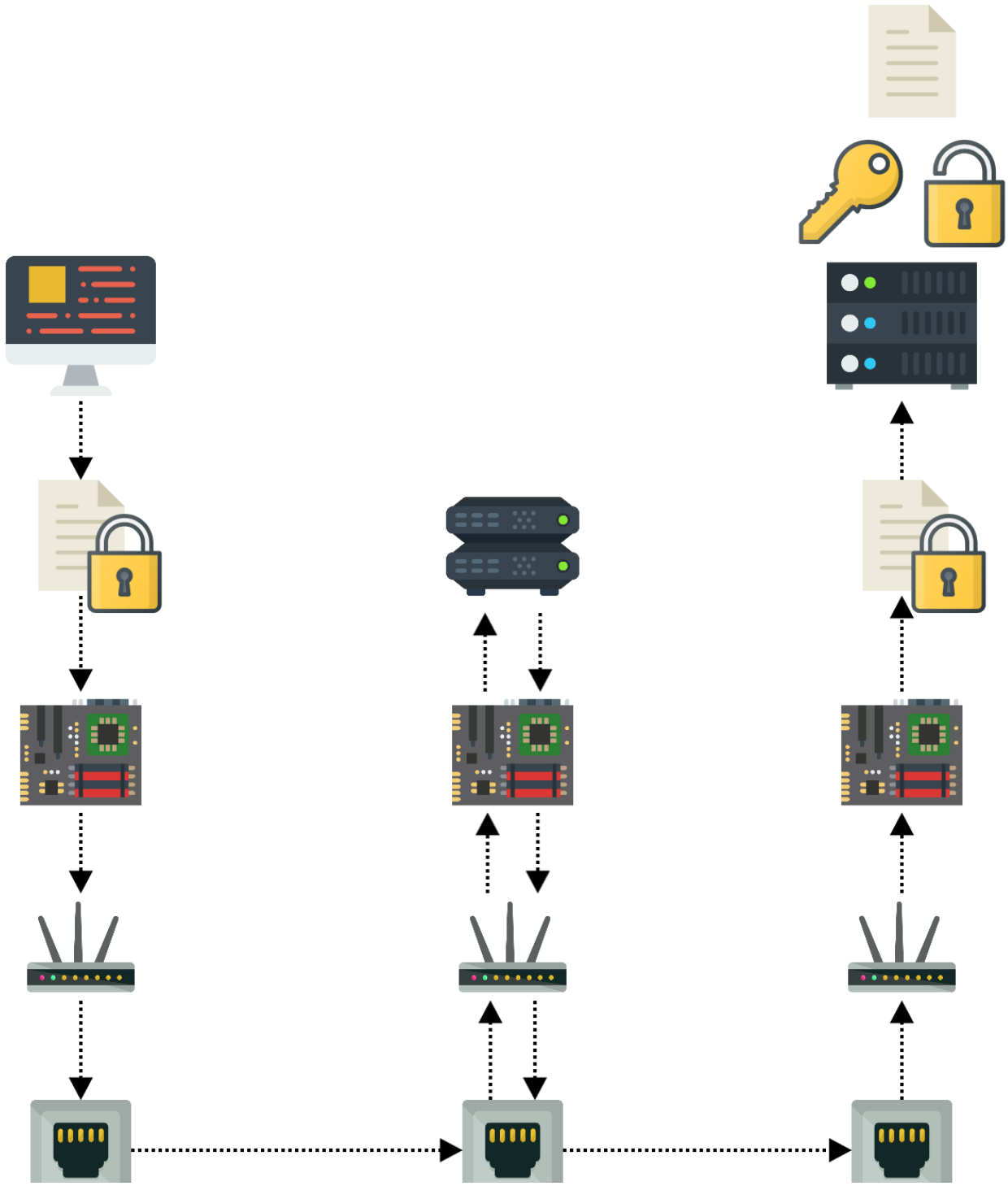
This approach lowers the resource utilization of the web server by offloading the heavy lifting of encrypting and decrypting network traffic onto the load balancer. The web server behind the load balancer only has to handle plaintext HTTP requests, as the requests have already been decrypted by the load balancer.

However, some applications have regulatory or security concerns that require that they have encryption of data while in transit. If you use a level 7 load balancer for such an application you would end up with this:



Because the application load balancer is operating at level 7 it has to decrypt the HTTP request to inspect its headers, and then encrypt the request again to send it to the client. Then your web server decrypts it again to read it. Not only does this add extra latency by doubling the SSL/TLS overhead, but it also means your private key has to be stored at the load balancer level as well as at the web server level.

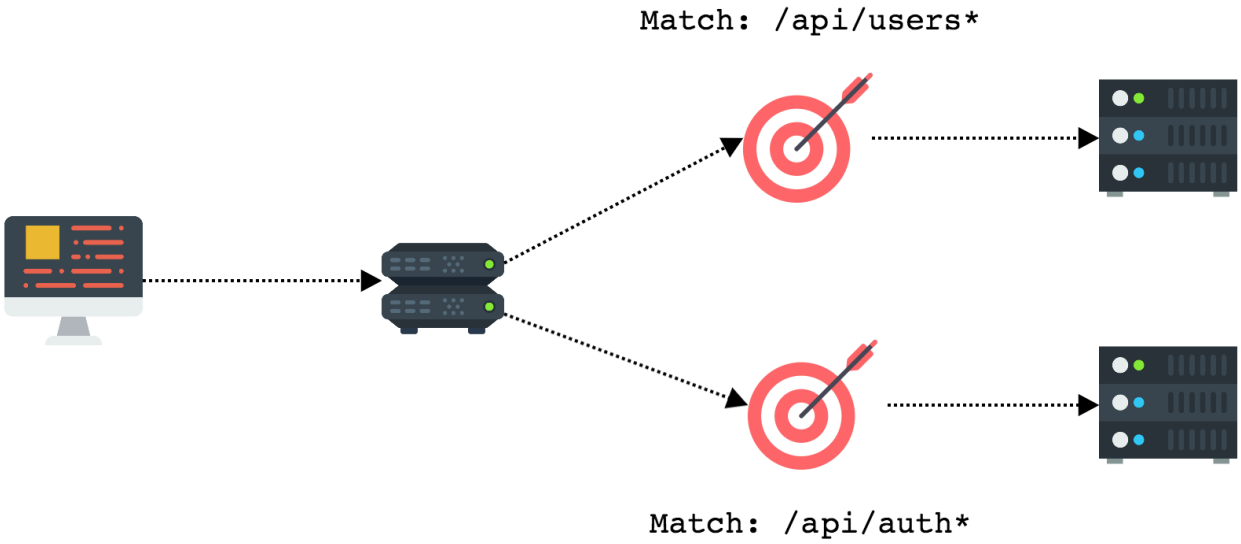
If end to end encryption is a requirement it would make more sense to use a level 4 network load balancer instead of a level 7 load balancer:



In this design the encrypted payload is not decrypted at the load balancer level. Instead the load balancer just directs the TCP packets to your web server. This reduces the latency, as well as giving the application true end-to-end encryption between client and web server.

HTTP Host and Path Based Routing

If end to end encryption is not needed then it is possible to take advantage of one of the powerful features of a level 7 application load balancer: host and path based routing.



In this configuration the application load balancer has rules that check the HTTP headers of an incoming request. If the request path matches one pattern then the application load balancer directs the request to one machine, but if it matches another pattern it directs the request to a different machine. This can be extremely useful for microservice or other distributed architectures where you have one public API, but multiple components powering different sections of the API.

Once again this type of routing can only be done using a load balancer at level 7 because it requires reading details of the HTTP request. The level 4 network load balancer only has TCP packets to work with, so it can't read the HTTP request headers like the application load balancer can.

Protocols other than HTTP

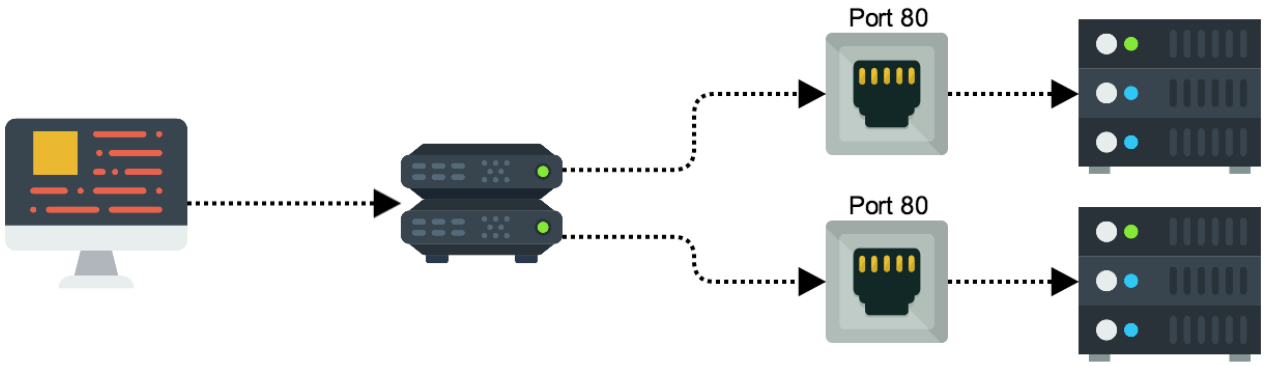
Not every web application is built on top of HTTP. For example, a realtime multiplayer game might implement its own lightweight protocol on top of TCP (or even UDP) for

communication between a game client and a game server. For such communication a layer 7 application load balancer would not work, because a load balancer at this level expects that all network traffic be HTTP.

A level 4 network load balancer should be used, as it can forward the TCP packets from the game client to the game server. In the case of a realtime game the level 4 network load balancer will also introduce less latency because network communication does not have to go all the way up and back down the network stack at the load balancer.

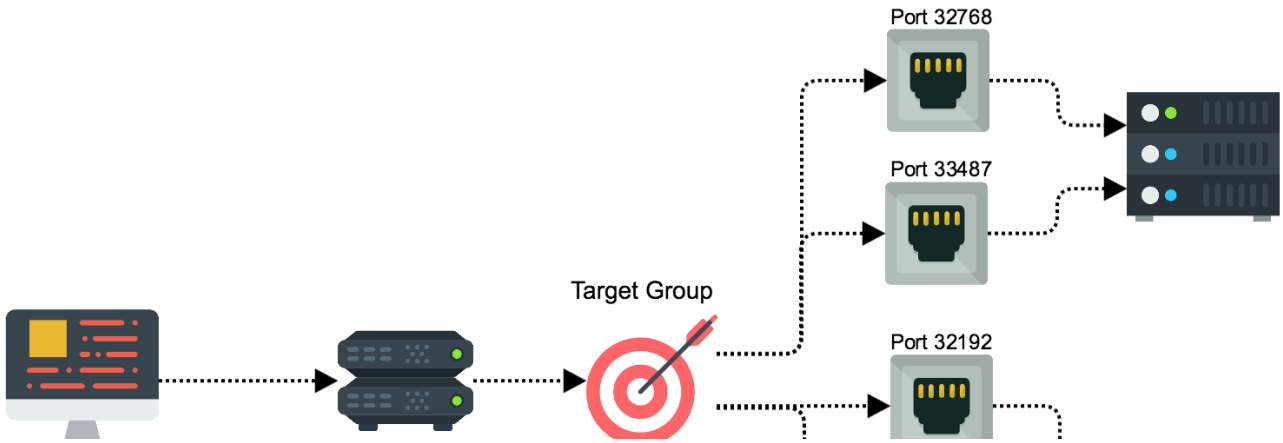
Dynamic Ports in ALB and NLB

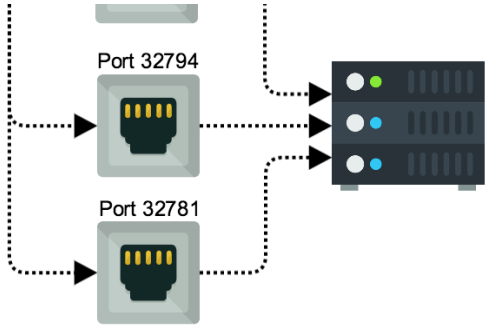
One feature that is common to both the ALB and the NLB is the ability to send network traffic to dynamic ports. In the past a load balancer traditionally looked like this:



The two backend instances behind the load balancer would each run an identical copy of the application running on a well known static port number. The load balancer would evenly distribute any requests it received to the same port number on each instance.

But with the rise of docker containers and binpacking strategies for running multiple applications on a single instance there are many architectures that look like this:

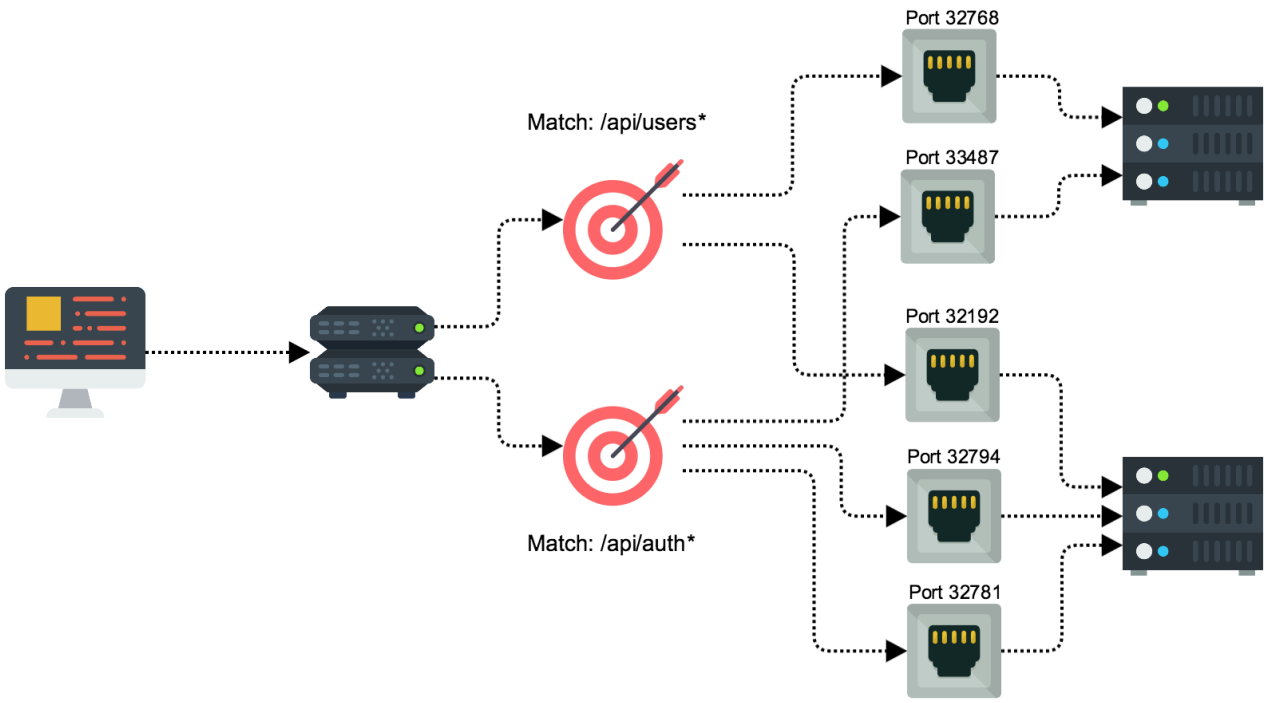




In this approach a single EC2 instance could be running multiple containers, each of which has been randomly assigned a port. These port numbers are not static. If the application containers are shutdown and restarted they may get a different port number assigned to them.

Both Application Load Balancer and Network Load Balancer support this new pattern of dynamic ports by using an AWS resource called the “Target Group”. A target group tracks the list of ports that are accepting traffic on each instance and gives the load balancer a way to distribute traffic evenly across ports. So in the above example 2/5ths of the traffic would be sent to the first instance that has containers hosted on two ports, while the remaining 3/5ths of the traffic would be sent to the other instance which has three open ports.

This also applies to an application load balancer that is doing HTTP based routing on level 7 of the network stack:



With this configuration the application load balancer will distribute 1/2 of all traffic that matches the route "/api/users*" to each of the two ports in one target group, while distributing 1/3 of all traffic that matches the route "/api/auth*" to each of the three ports in the other target group.

Integrating ALB and NLB with EC2 Container Service

Because of their dynamic port support the second generation load balancers are ideal for routing traffic to containerized services. There is a seamless integration between ALB and NLB, and Amazon EC2 Container Service (ECS). ECS is Amazon Web Service's managed orchestration system for deploying and operating docker containers across a fleet of instances. It is designed to provide an easy way to connect the broad ecosystem of AWS services to containers.

Deploying a containerized application behind an ALB or NLB starts with the application itself. For example, this is a basic Node.js web application:

```
const app = require('koa')();
const router = require('koa-router')();

router.get('/', function *() {
  this.body = 'Ready to receive requests';
});

app.use(router.routes());
app.use(router.allowedMethods());

app.listen(8081);
```

This application binds to port 8081, and expects all traffic to arrive on that port. If we were deploying this application without containers it would be run on several instances, and a load balancer would route all traffic to the static port 8081 on each instance.

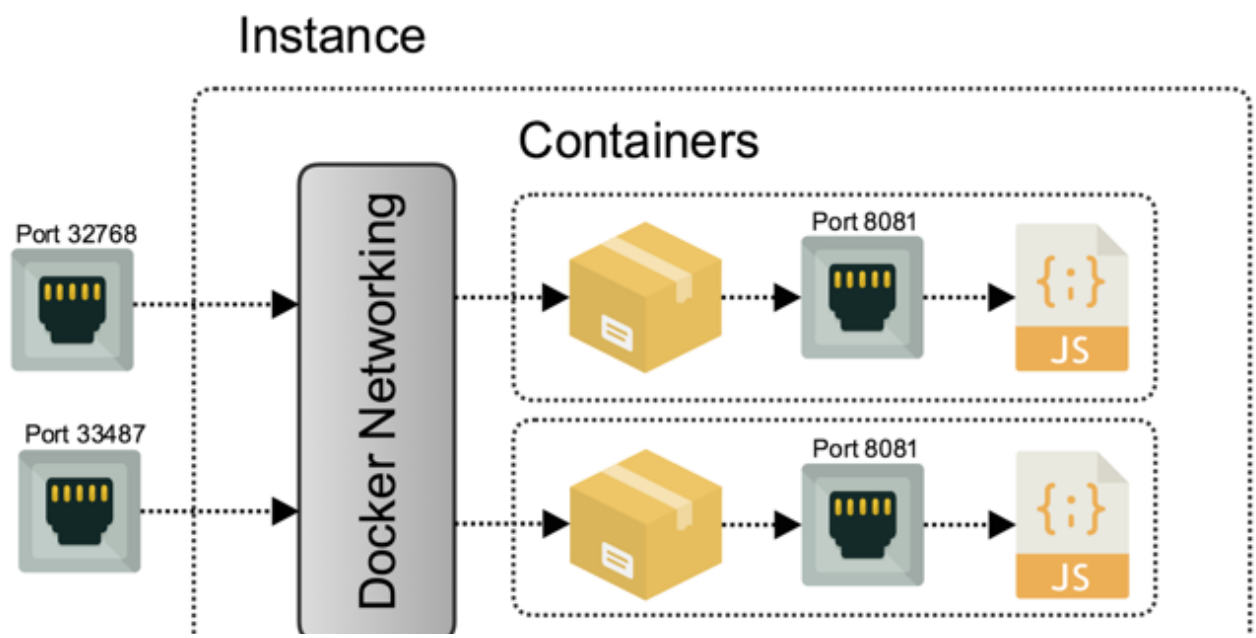
But because we are deploying this application inside a docker container it allows us to take advantage of the docker feature of mapping ports inside a container to different ports on the host which is running the container. This is accomplished using a simple configuration that is passed to docker on container launch. In ECS we use a resource type called a **task definition** to define this configuration. The task definition is a lightweight metadata document that tells ECS what parameters to launch your docker container with.

Here is an example of a task definition that we would use to run a container for the sample application above:

```
{
  "family": "api",
  "containerDefinitions": [
    {
      "name": "api",
      "image": "000000000000.dkr.ecr.us-east-2.amazonaws.com/api:v1",
      "cpu": 256,
      "memory": 256,
      "portMappings": [
        {
          "hostPort": 0,
          "containerPort": 8081,
          "protocol": "tcp"
        }
      ]
    }
  ]
}
```

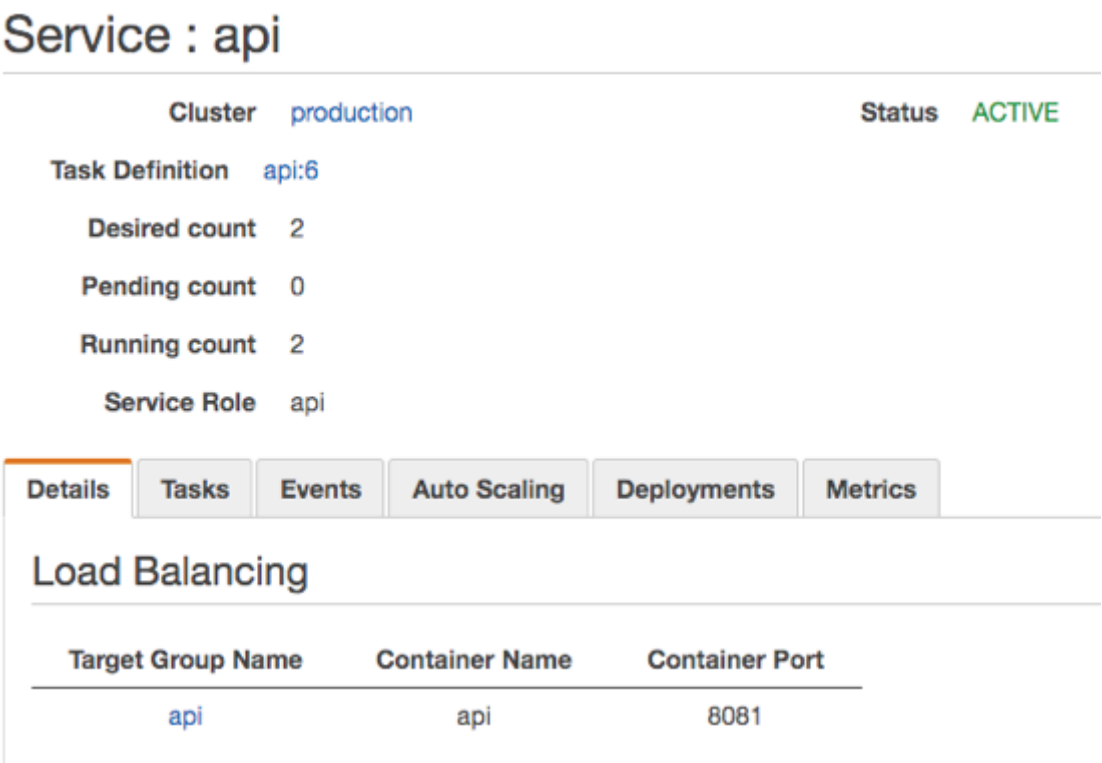
You can see that the container port is specified to be 8081, which is the same port that the application code is binding to. However, the host port is set to zero. This tells docker that we do not want a specific static port on the host. Instead docker can bind to a random port on the host, and forward traffic that arrives on that port to port 8081 inside the container.

With this configuration multiple instances of a container can be deployed onto a single machine:

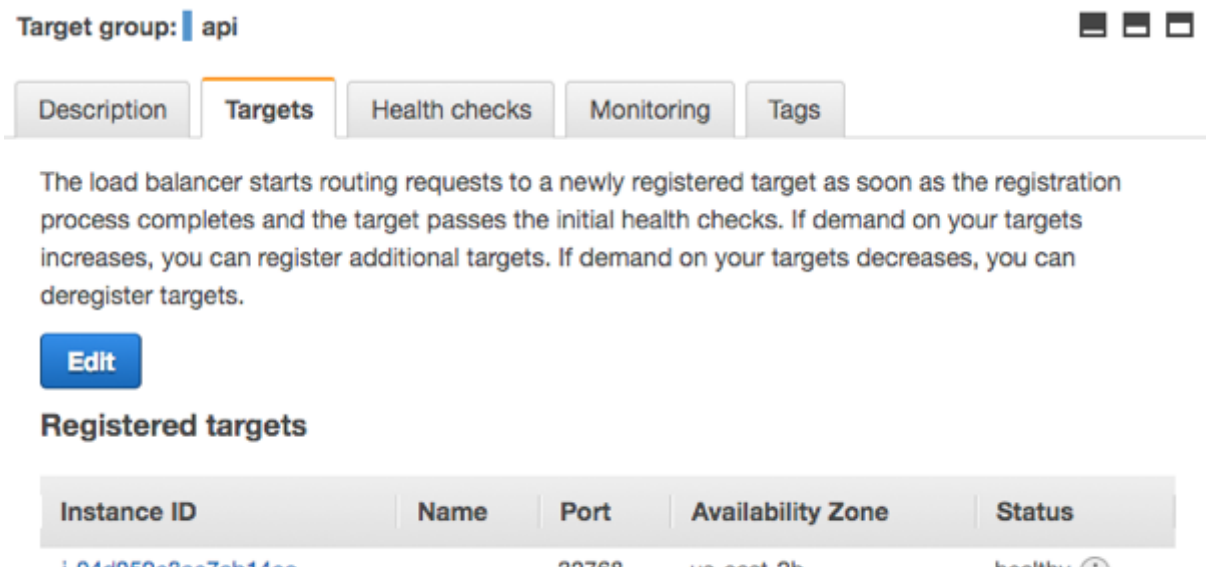


Any traffic arriving on port 32768 is forwarded to port 8081 in one container, while any traffic arriving on port 33487 is forwarded to port 8081 in the other container.

The task definition with its dynamic port mapping configuration can be used to launch a service in EC2 Container Service:

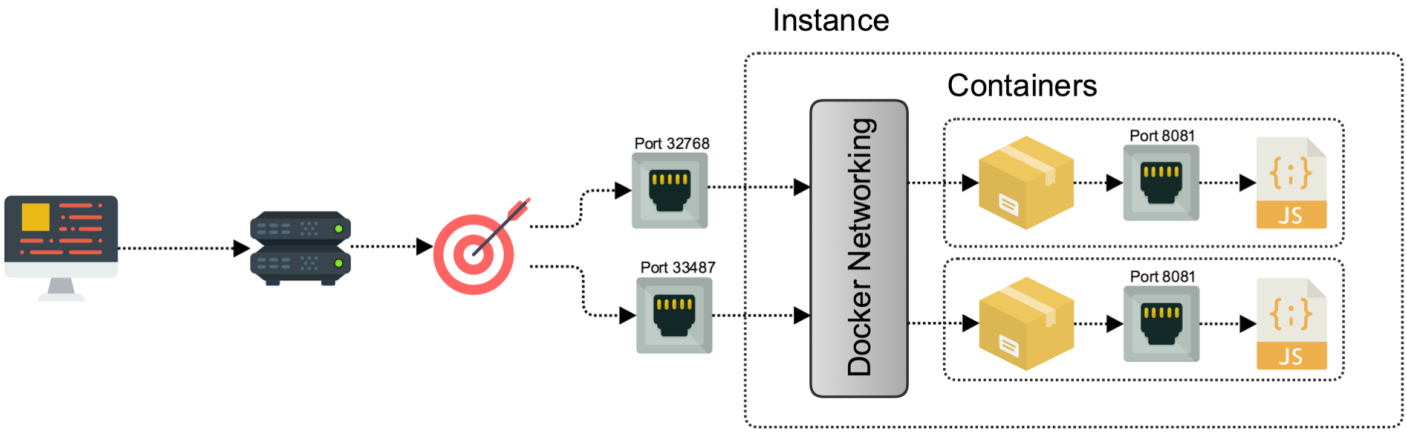


Each ECS service that is connected to a load balancer comes with a target group that is automatically kept in sync as new tasks are launched, or replaced. The target group has a list of each instance and the ports that are accepting traffic on that instance:



i-04d959e3ae7ab14ec	32768	us-east-2b	healthy
i-04d959e3ae7ab14ec	33487	us-east-2b	healthy

An ALB or NLB connected to the target group can use this list to pick an instance and port to send network traffic to. The full pipeline looks like this:



1. A client application initiates a new connection to the load balancer.
2. The load balancer receives the traffic, and picks a target from the target group attached to the load balancer. The target group has been configured by EC2 Container Service with a list of instances and ports.
3. The load balancer sends the traffic to the chosen instance and port.
4. The docker networking layer accepts the traffic and forwards it to the configured port inside the right container.
5. Your application running inside the container receives the traffic on the port it has bound to.

Conclusion

Both Application Load Balancer and Network Load Balancer are designed from the ground up for the modern paradigm of dynamic port configurations as commonly seen in containerized deployments. Picking which load balancer is right for you will depend on the specific needs of your application, such as whether or not network traffic is HTTP, whether you need end to end SSL/TLS encryption, and whether or not you want host and path based traffic routing.

If you are deploying docker containers and using a load balancer to send network traffic to them EC2 Container Service provides a tight integration with ALB and NLB so you can keep your load balancers in sync as you start, update, and stop containers across your fleet.

Amazon Web Services Microservices Software Architecture AWS Networking