

BandTec

DIGITAL SCHOOL

Computação em Nuvem

Sistemas Distribuídos

Sockets

Professor: Rogério Chola

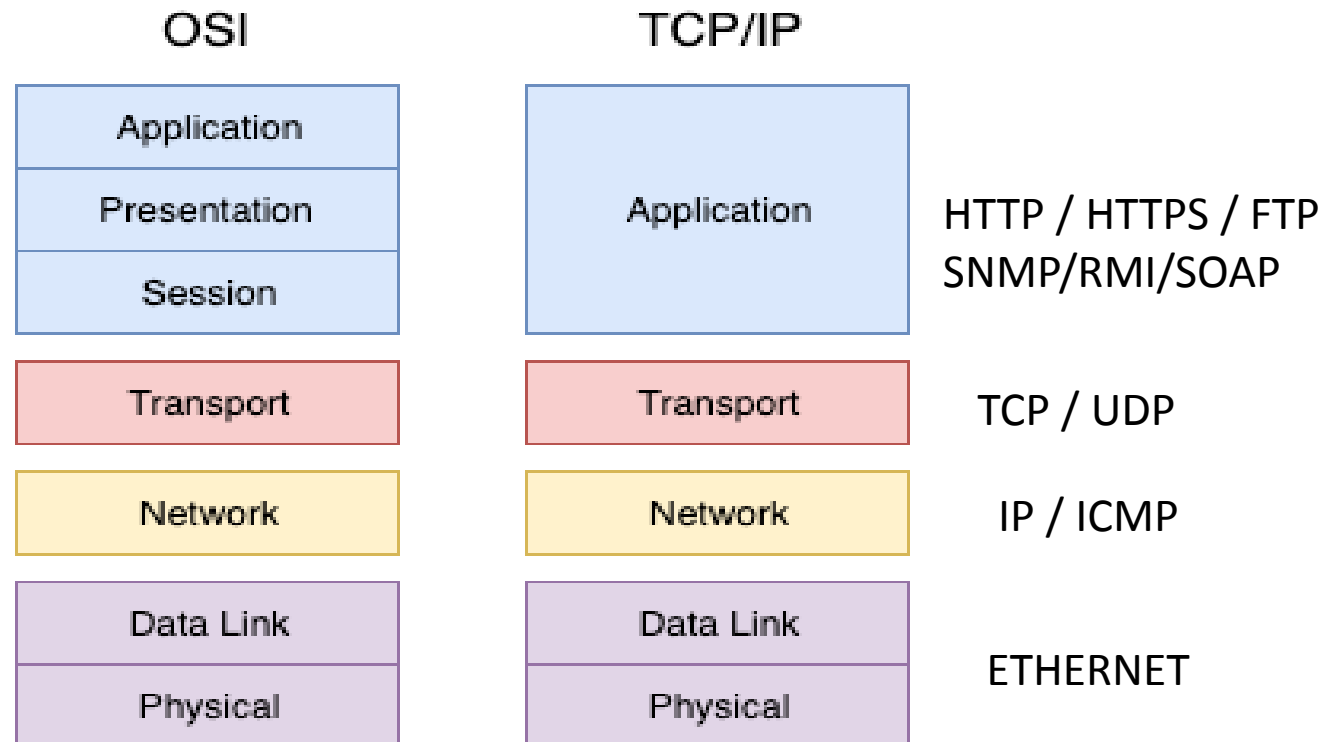
e-mail: rogerio.chola@bandtec.com.br

Uma rede de computadores caracteriza-se por dois ou mais computadores interligados, não importando por qual meio (cabos, ondas de rádio etc), desde que sejam capazes de trocar informações entre si e/ou compartilhar seus recursos de hardware

Em uma rede de computadores também podemos ter clientes e servidores. *Servidor* é uma máquina que fornece um serviço qualquer na rede e *Cliente* é uma máquina que consome o serviço fornecido pelo *Servidor*. Também dizemos que algumas aplicações são clientes e outras servidoras, quando fornecem ou consomem serviços na rede. Por exemplo, o [Mozilla Thunderbird](#) é um cliente de e-mail, já o [Postfix](#) é um servidor de e-mail. Um navegador de internet (Chrome, Firefox etc) é uma aplicação cliente que requisita dados de um servidor. Uma rede funciona sob protocolos (que são regras que definem o funcionamento dela) e a família de protocolos mais conhecida e utilizada é a **TCP/IP**, que engloba os protocolos *IP* (da camada de rede), *TCP* e *UDP* (da camada de transporte), *HTTP* (da camada de aplicação) entre outros

A referência de como as redes funcionam e são construídas vem do [modelo OSI](#), que foi definido pela ISO em meados dos anos 80 para servir de referência para o projeto de hardware e software das redes, pois o que acontecia é que cada fornecedor implementava o seu próprio padrão e isso prejudicava a interoperabilidade. Apesar do modelo OSI ainda ser a referência teórica (o modelo) para as redes, a arquitetura **TCP/IP** é que tem a aplicabilidade e o protagonismo nas intranets e na internet

O modelo OSI possui sete camadas enquanto o TCP/IP é dividido em cinco ou quatro. O diagrama abaixo mostra onde as camadas do modelo TCP/IP se “encaixam” no OSI:



O protocolo IP (Internet Protocol) é o mais importante da família TCP/IP (Transporte/Rede) e deve ser associado com outros protocolos sendo os mais importantes e mais utilizados o TCP e o UDP

TCP x UDP

Ambos são protocolos da camada de transporte e, quando precisamos de confiabilidade no transporte do dado, usamos o protocolo *IP* associado ao *TCP* (que garante a entrega das informações). Quando priorizamos mais velocidade e menos controle, associamos o protocolo *IP* ao *UDP* (tráfego de voz e vídeo são bons exemplos onde o UDP teria boa aplicabilidade, pois mesmo perdendo um ou outro pacote, não interfere totalmente no todo, permanecendo inteligível). Observe que a comunicação no TCP se dá nas duas pontas:



Algumas das principais características do TCP (*Transmission Control Protocol*):

- Orientado à conexão (só transmite dados se uma conexão for estabelecida depois de um *Three-way Handshake*);
- É Full-duplex, ou seja, permite que as duas máquinas envolvidas transmitam e recebam ao mesmo tempo;
- Garante a entrega, sequência (os dados são entregues de forma ordenada), não duplicação e não corrompimento;
- Automaticamente divide as informações em pequenos pacotes (fragmentação);
- Garante equilíbrio no envio dos dados (para não causar “sobrecarga” na comunicação);

TCP x UDP

No UDP, a comunicação se dá em uma ponta e, se algum segmento falha, isso é ignorado e o fluxo continua:



Algumas das principais características do UDP (*User Datagram Protocol*):

- Diferente do TCP ele **não** é orientado à conexão;
- Não é confiável como o TCP, ele não garante a entrega completa dos dados;
- É preciso que dividamos manualmente os dados em datagramas (entidades de dados);
- Não garante a sequência da entrega, portanto, os dados podem chegar em uma ordem aleatória;

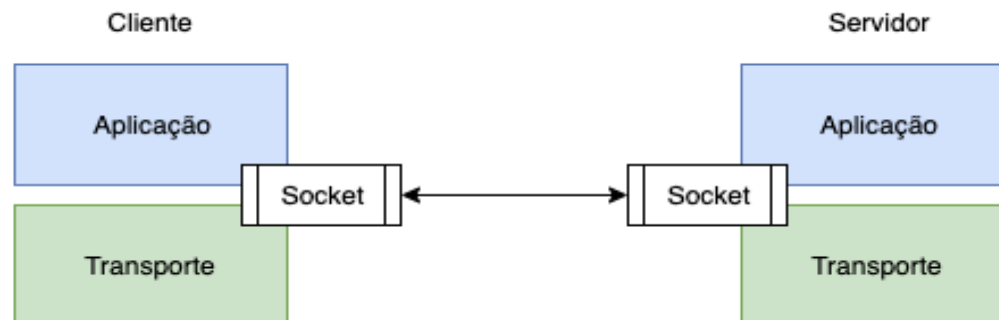
Por essas características menos restritivas (menos controles) ele é muito mais rápido que o TCP. Como contrapartida, ele exige um pouco mais do desenvolvedor na hora de implementá-lo. Por fim, é importante pontuar que tanto o *UDP* quanto o *TCP* trabalham com portas, que são elementos lógicos que interligam clientes e servidores de aplicações em redes *TCP/IP*. O cliente precisa saber qual porta ele se conectará no servidor. Por exemplo, servidores web por padrão usam a porta *80* para servir as páginas e quando acessamos uma página web usando o protocolo *http* uma conexão *TCP* à porta *80* do servidor é feita

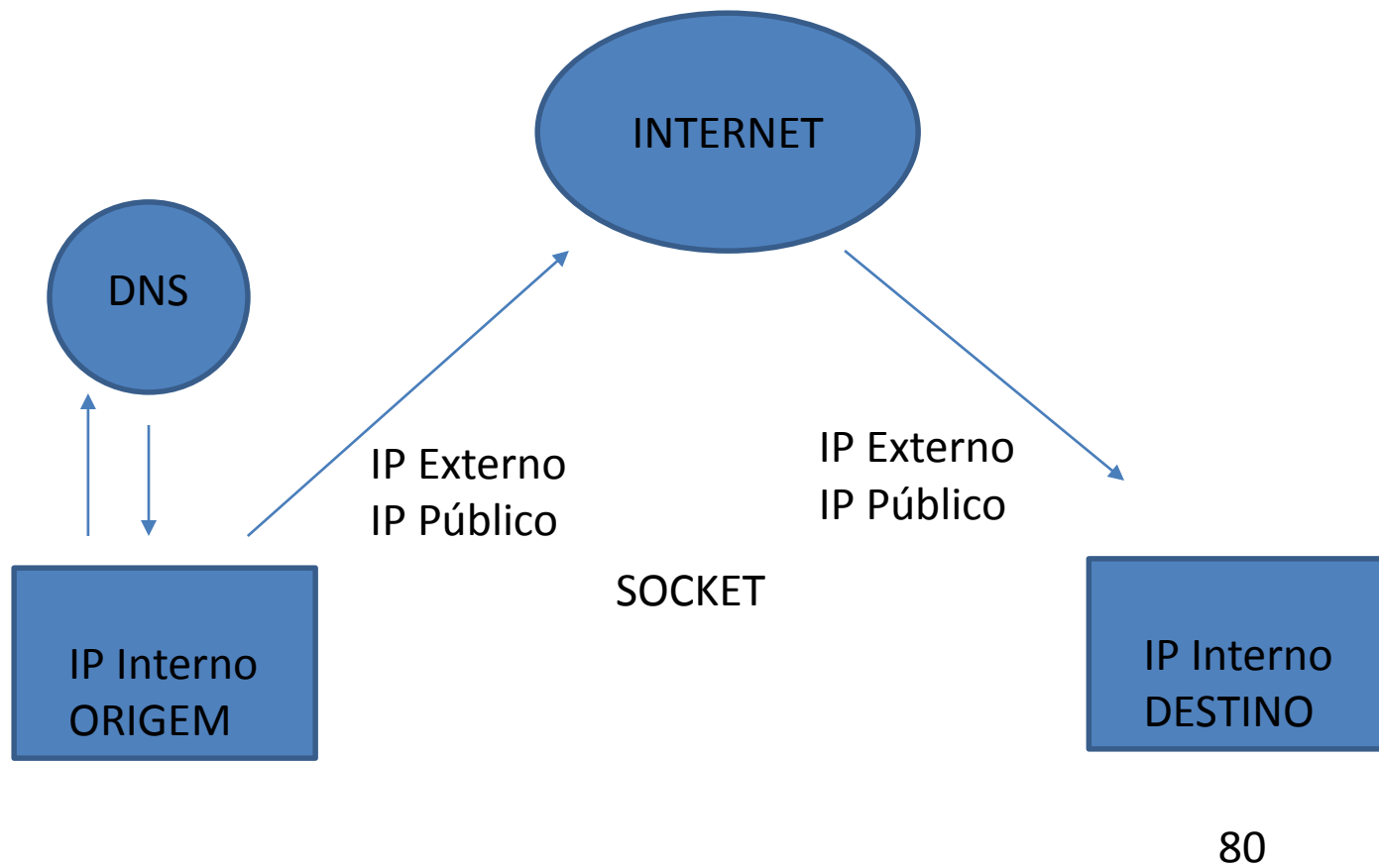
Socket é o elemento que provê a comunicação entre duas pontas (origem e destino) – também conhecido como two-way communication – entre dois processos que estejam na mesma máquina (Unix Socket) ou na rede (TCP/IP Sockets). Na rede, a representação de um socket se dá por **IP:PORTA**

Por exemplo: 127.0.0.1:4477 (IPv4). Um socket que usa rede é um Socket do tipo TCP/IP.

Muito do que fazemos no dia a dia faz uso de sockets. O nosso navegador utiliza sockets para requisitar as páginas; quando acessamos o nosso servidor pelo protocolo de aplicação SSH também estamos abrindo e utilizando um socket.

Sabendo que TCP/IP é base da nossa comunicação na internet, considerando o modelo de rede OSI, os sockets estão entre a camada de aplicação e a de transporte. No modelo de endereçamento IPv6 os Sockets podem estabelecer uma conexão fim-a-fim

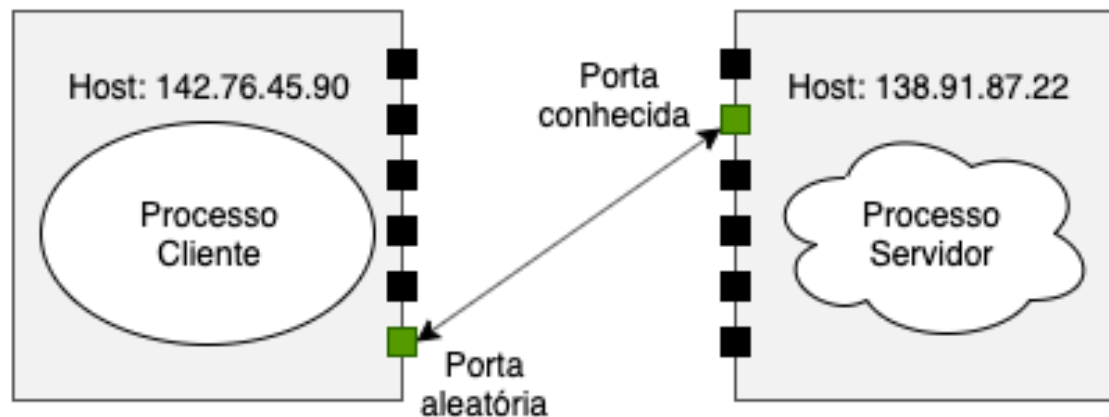




Para os processos envolvidos a sensação é que a comunicação está acontecendo diretamente entre eles, no entanto, ela está passando pelas camadas da rede. Essa abstração provida pelos Sockets é o que chamamos de comunicação lógica. Outra forma de entender os Sockets é que eles são a “interface” de comunicação inter-processos

Todo cliente deve conhecer o *socket* do servidor (conjunto ip e porta) para se comunicar, mas o servidor só vai conhecer o *socket* do cliente quando este realizar uma conexão com ele, ou seja, a conexão no modelo *cliente-servidor* é sempre iniciada pelo cliente

O diagrama abaixo mostra que a porta do servidor precisa ser previamente conhecida pelo cliente, enquanto que para o servidor não importa qual é a porta do cliente, ele vai conhecê-la quando a conexão dele com o cliente for estabelecida



Sistemas operacionais baseados no Unix proveem uma interface padrão para operações *I/O* (input e output) que se passa por descritores de arquivo

Um descritor de arquivo é representado por um número inteiro que se associa a um arquivo aberto e, nesses sistemas, há uma generalização de que “tudo é um arquivo”, então, nesse contexto, um arquivo pode ser uma conexão de rede (um socket é um tipo especial de arquivo), um arquivo de texto, um arquivo de áudio, até mesmo uma pasta é um tipo especial de arquivo

Como um socket se comporta como um arquivo, chamadas de sistema de leitura e escrita são aplicáveis, da mesma forma como funcionam em um arquivo ordinário, e é aqui que entra a programação de sockets com a API POSIX sockets

Linguagens como o Java, PHP, Perl, Python, etc, abstraem isso fornecendo ao desenvolvedor uma API de ainda mais alto nível

Unix Socket

Em sistemas Unix e agora recentemente também no [Windows 10](#), temos um mecanismo para a comunicação entre processos que estão no mesmo host (ao invés da rede), chamado de Unix Socket

A diferença entre um [Unix Socket](#) (IPC Socket) de um [TCP/IP Socket](#) é que o primeiro permite a comunicação entre processos que estão na mesma máquina. Já o segundo, além disso, permite a comunicação entre processos através da rede

No entanto, um TCP/IP Socket também pode ser usado para a comunicação de processos que estão na mesma máquina através do [loopback](#) que é uma interface virtual de rede que permite que um cliente e um servidor no mesmo host se comuniquem como se fosse uma comunicação remota. No caso de IPv4 via IP 127.0.0.1 e IPv6 ::1

A particularidade é que Unix Sockets estão sujeitos às permissões do sistema e costumam ser um pouco mais performáticos, pois não precisam realizar algumas checagens e operações, por exemplo, de roteamento, algo que acontece com os TCP/IP Sockets. Ou seja, se os processos estão na mesma máquina, Unix Sockets podem ser a melhor opção, mas se estiverem distribuídos na rede, os TCP/IP Sockets são a escolha correta

Se você tem acesso a algum servidor baseado em Unix/Linux, execute `netstat -a -p --unix` que serão listados todos os Unix Sockets abertos no sistema operacional (bem como mostrará outras informações como o tipo do Socket, caminho etc). Para visualizar tanto os TCP/IP Sockets quanto os Unix Sockets, execute um `netstat -natp` que listará todos os sockets abertos:

```
Active Internet connections (servers and established)
```

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.1:9070	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:9072	0.0.0.0:*	LISTEN

```
...
```

```
Active UNIX domain sockets (servers and established)
```

Proto	RefCnt	Flags	Type	State	I-Node	Path
unix	2	[]	DGRAM		26007836	/run/user/1000/systemd/notify
unix	2	[ACC]	SEQPACKET	LISTENING	1976	/run/udev/control

```
...
```

Quanto ao TIPO (Type) existem quatro tipos de sockets: Stream Sockets, Datagram Sockets, Raw Sockets e Sequenced Packet Sockets sendo que os dois primeiros são os mais comuns e utilizados:

Stream Sockets (SOCK_STREAM): Esse tipo usa o **TCP**, portanto, todas as características enumeradas anteriormente se aplicam a ele: garantia de entrega e ordem, orientado à conexão etc;

Datagram Sockets (SOCK_DGRAM): Esse tipo usa o **UDP**, portanto, não é orientado à conexão, não garante entrega completa dos dados e exige um controle mais especial do desenvolvedor

Sockets estão presentes em quase tudo o que fazemos na internet, naquele jogo multiplayer que você joga, naquele chat que você iniciou online e muito mais

As linguagens de programação (ou extensões delas) abstraem grande parte da programação com sockets

Estilos de Arquitetura

A forma como os componentes são configurados é definida como sendo o estilo arquitetural do sistema.

Vejamos alguns desses estilos: Arquitetura de Objetos; Arquitetura cliente-servidor; Arquitetura em Camadas

Como os componentes são distribuídos

Em geral existem, 3 abordagens: Arquiteturas Centralizadas; Arquiteturas Híbridas; Arquiteturas Descentralizadas

Arquiteturas Centralizadas são comumente referidas como cliente servidor, onde vários clientes solicitam serviços a um servidor. Claro que um servidor pode ser cliente de um outro **servidor** que ofereça um serviço.

Evolução das arquiteturas

1 Camada

Todo o processamento centralizado no mainframe (apresentação, logica de negocio, persistência de dados). Dominantes até a década de 80. Terminais burros apenas apresentavam os dados

2 Camadas

Vantagens: Também conhecida como cliente servidor. Surgiu nos anos 90. Composta pela camada cliente e a camada servidor. Algum processamento local aproveitando capacidade dos PC; Melhor interface com o usuário; Libera o servidor de validação de entrada de dados e outros processamentos que podem ser feitos no cliente.

Desvantagens: Enormes problemas de atualização e manutenção da camada cliente; Escalabilidade limitada; Cada cliente com uma conexão com o servidor

Evolução das arquiteturas

3 Camadas

Composta pela camada de apresentação, camada de logica de negocio e camada de dados. Muito flexível podemos trocar uma camada sem afetar a outra camada. Problemas de atualização e manutenção da camada cliente são reduzidos drasticamente. Aumento da escalabilidade e confiabilidade.

N Camadas

Composta por várias camadas que são conectadas via protocolos da web (http, shttp, soap, RMI, etc.). Melhor modularização da aplicação com camadas com responsabilidades bem definidas. Compartilhamento de serviços e reaproveitamento de código (SOA – Service Oriented Architecture)

Desafios: maior complexidade de implementação (tolerância a falhas, transações distribuídas, pool de conexões de banco de dados, balanceamento de carga); Maior esforço para administração do ambiente (aplicação, banco de dados, etc.); Maior flexibilidade possível; Alto custo com infraestrutura. Em geral sistemas distribuídos são altamente complexos

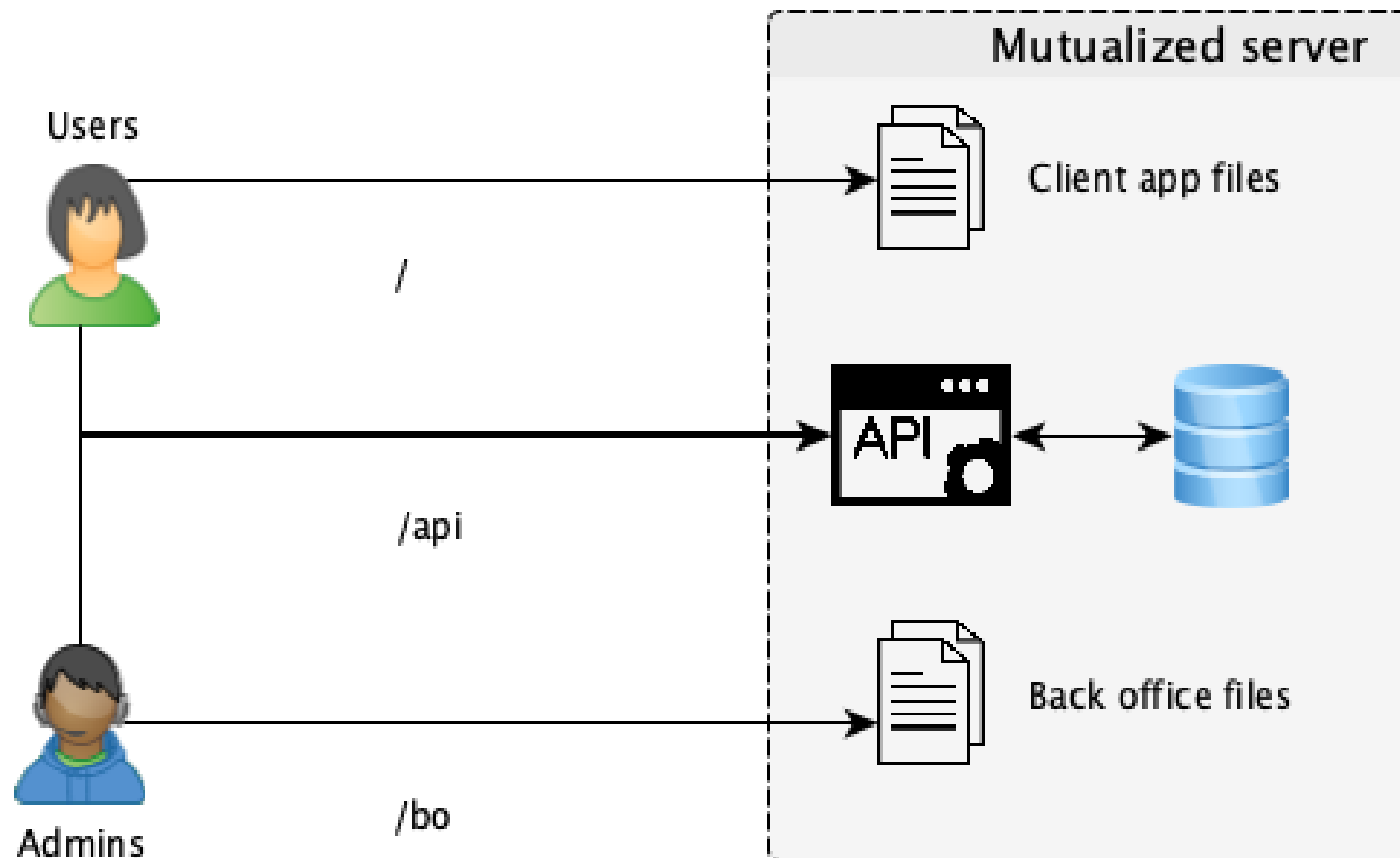
Instância de aplicação executando em máquina virtual;
Atendimento via API – Stateless e RESTfull – acessível via
chamadas HTTP/HTTPS

Escalabilidade – Scale Up

Não-blocante

Simplicidade

Versões User/Admin



NGINX

Pronunciado “engine-ex,” é um famoso software de código aberto para servidores web lançado originalmente para navegação HTTP. Atualmente, também é muito utilizado como proxy reverso, balanceador de carga HTTP, e proxy de email para os protocolos IMAP, POP3, e SMTP.

O NGINX foi lançado em Outubro de 2004. O criador do software, Igor Sysoev, começou o projeto em 2002 como uma resposta ao problema C10k. O C10k é o desafio de gerenciar 10 mil conexões ao mesmo tempo. Atualmente há ainda mais conexões que um servidor gerencia. Por este motivo o NGINX oferece uma arquitetura orientada a eventos e assíncrona, o que o torna um dos servidores mais confiáveis em questão de velocidade e escalabilidade.

NGINX

Devido a sua habilidade de suportar muitas conexões com alta velocidade, muitos sites de alto tráfego tem utilizado o NGINX, como Google, Netflix, Adobe, Cloudflare, WordPress.com, e muitos outros.

NGINX – Funcionamento Básico

Sempre que um usuário faz uma solicitação de carregamento de página o navegador entra em contato com o servidor do site. Então o servidor busca pelos arquivos solicitados e os entrega ao navegador. Esse é o tipo de solicitação mais simples.

O exemplo acima é considerado uma thread individual. Servidores web tradicionais criam uma thread individual para cada solicitação, mas o NGINX não funciona assim. No caso o NGINX performa com uma arquitetura assíncrona e orientada a eventos. Isso significa que threads similares são gerenciadas por um worker process, e cada worker process contém unidades menores chamadas conexões worker. Esta unidade inteira então é responsável por cuidar das solicitações de threads. As conexões worker levam as solicitações até um processo worker, que por sua vez as envia para o processo master. Finalmente o processo master fornece o resultado da solicitação.

NGINX – Funcionamento Básico

Isso pode parecer simples, mas uma única conexão *worker* pode cuidar de até 1024 solicitações similares. Por isso o NGINX consegue atender milhares de solicitações sem dificuldades. Também é o motivo de o NGINX ter se tornado uma excelente opção para websites com muito movimento como e-commerces, mecanismos de busca e armazenamento em cloud.



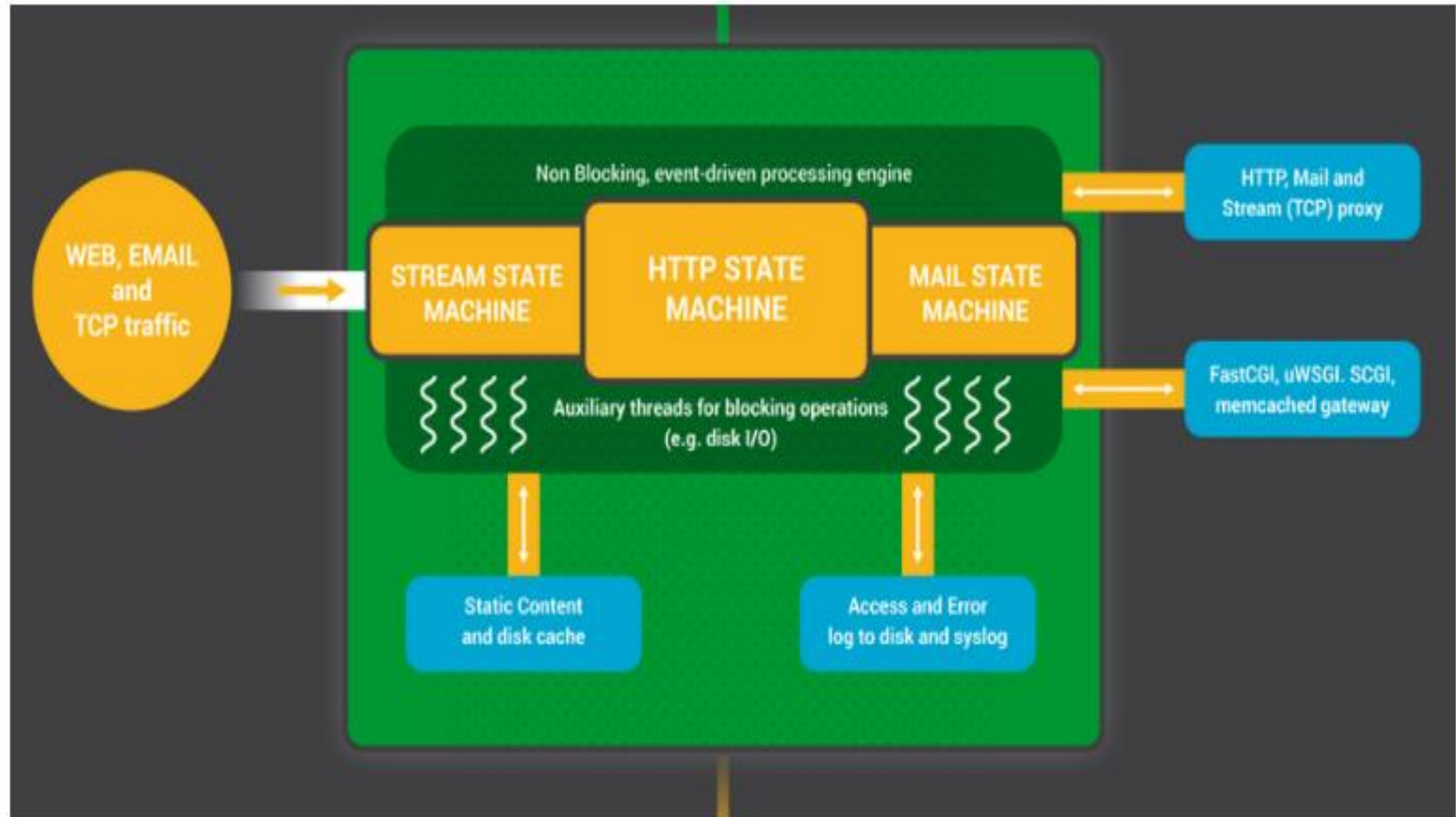
How Does NGINX Work?

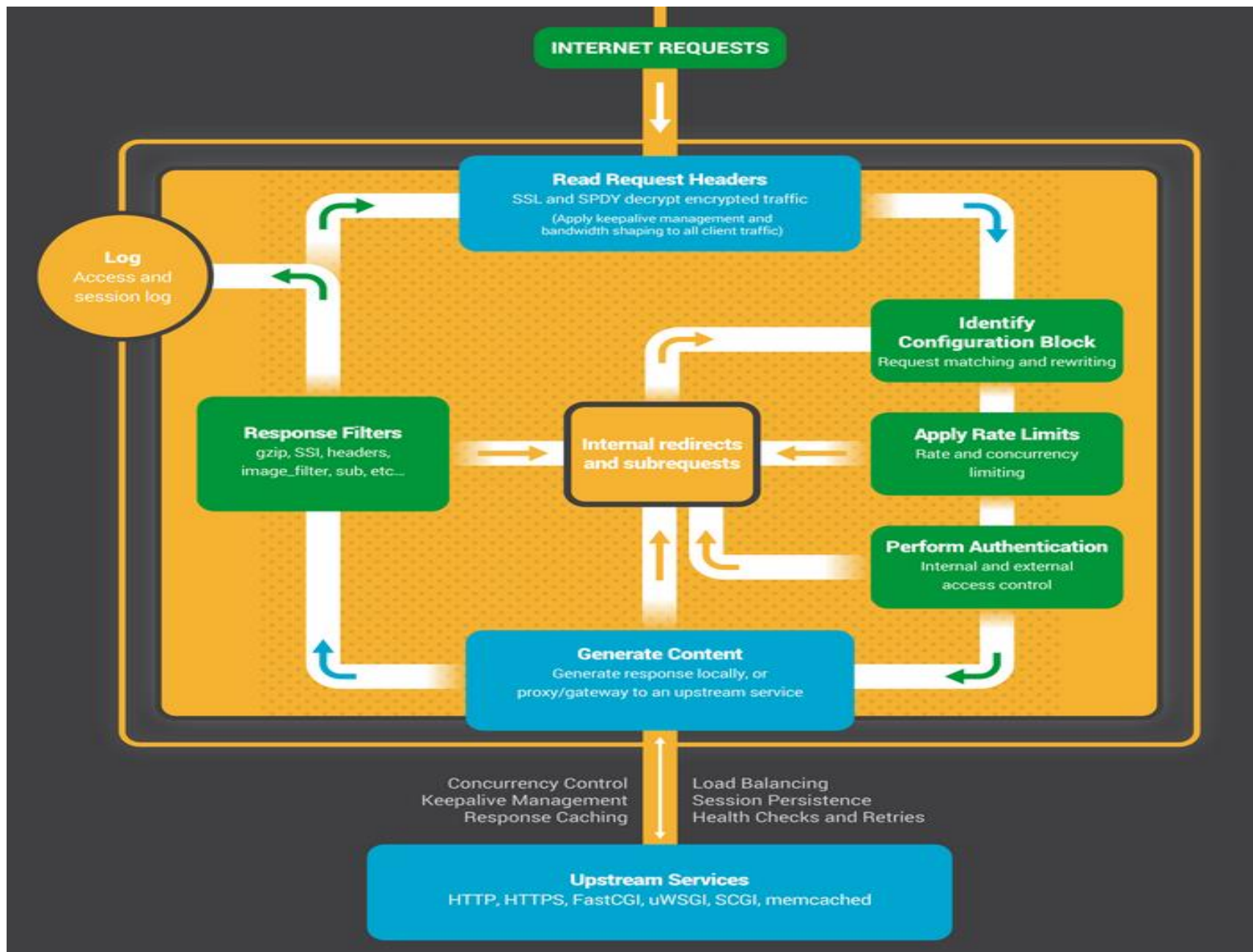
NGINX uses a predictable process model that is tuned to the available hardware resources:

- The *master* process performs the privileged operations such as reading configuration and binding to ports, and then creates a small number of child processes (the next three types).
- The *cache loader* process runs at startup to load the disk-based cache into memory, and then exits. It is scheduled conservatively, so its resource demands are low.
- The *cache manager* process runs periodically and prunes entries from the disk caches to keep them within the configured sizes.
- The *worker* processes do all of the work! They handle network connections, read and write content to disk, and communicate with upstream servers.

The NGINX configuration recommended in most cases – running one worker process per CPU core – makes the most efficient use of hardware resources. You configure it by setting the **auto** parameter on the `worker_processes` directive:

Inside the NGINX Worker Process





Most web application platforms use blocking (waiting) I/O

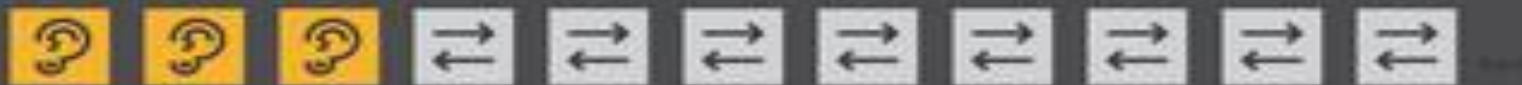
Listen Sockets (port 80, 443, etc)



Each worker can only process one active connection at a time





NGINX uses a Non-Blocking “Event-Driven” architecture

Listen Sockets & Connection Sockets







Wait for an event (epoll or kqueue)

Event on Listen Socket:

- accept  new 
- set  to be non-blocking
- add  to the socket list

Event on Connection Socket:

- data in read buffer? read 
- space in write buffer? write 
- error or timeout? close  & remove  from socket list

An NGINX worker can process hundreds of thousands of active connections at the same time

NGINX vs APACHE

Dentre os servidores web mais populares o Apache é um dos principais rivais do NGINX. Ele existe desde a década de 90 e possui uma grande comunidade de usuários.

Compatibilidade de SO

Compatibilidade é um dos detalhes que devem ser levados em consideração ao escolher um servidor. Tanto o NGINX quanto Apache conseguem operar em muitos sistemas operacionais que suportam o sistema UNIX. Infelizmente a performance do NGINX no Windows não é tão boa como em outras plataformas.

Suporte ao Usuário

Usuários, desde iniciantes até profissionais, sempre precisam de uma comunidade para ajudar quando um problema surgir. Enquanto que ambos NGINX e Apache possuem suporte via email e um fórum no Stack Overflow, o Apache deixa a desejar no suporte vindo da própria empresa, a Apache Foundation.

NGINX vs APACHE

Performance

NGINX consegue executar 1000 conexões de conteúdo estático simultaneamente com o dobro de velocidade do Apache utilizando menos memória. Em relação à execução de conteúdos dinâmicos ambos possuem a mesma velocidade. NGINX é uma melhor opção para quem possui um site mais estático.

RESUMO

NGINX é um servidor web que também funciona como proxy de email, proxy reverso, e balanceador de carga. A estrutura do software é assíncrona e orientada a eventos; possibilitando o processamento de muitas solicitações ao mesmo tempo. O NGINX também é altamente escalável, significando que seu serviço cresce com o aumento de tráfego do usuário. NGINX e Apache são sem dúvidas dois dos melhores servidores web do mercado.

NGINX: Proxy Reverso vs Balanceador de Carga

Servidores proxy reversos e balanceadores de carga são componentes em uma arquitetura de computação cliente-servidor. Ambos atuam como intermediários na comunicação entre clientes e servidores, desempenhando funções que melhoram a eficiência. Eles podem ser implementados como dispositivos dedicados e criados especificamente, mas cada vez mais nas arquiteturas modernas da Web, são aplicativos de software executados em hardware comum.

Basicamente:

Um proxy reverso aceita uma solicitação de um cliente, a encaminha para um servidor que pode atendê-la e retorna a resposta do servidor ao cliente.

Um balanceador de carga distribui solicitações de entrada de clientes entre um grupo de servidores, retornando em cada caso a resposta do servidor selecionado para o cliente apropriado.

NGINX: Balanceador de Carga

SOCKETS

Os balanceadores de carga geralmente são implantados quando um site precisa de vários servidores, pois o volume de solicitações é muito alto para um único servidor lidar com eficiência. A implantação de vários servidores também elimina um único ponto de falha, tornando o site mais confiável. Geralmente, todos os servidores hospedam o mesmo conteúdo, e o trabalho do balanceador de carga é distribuir a carga de trabalho de uma maneira que faça o melhor uso da capacidade de cada servidor, evite a sobrecarga em qualquer servidor e resulte na resposta mais rápida possível ao cliente.

Um balanceador de carga também pode aprimorar a experiência do usuário, reduzindo o número de respostas de erro que o cliente vê. Ele faz isso detectando quando os servidores ficam inativos e desviando as solicitações deles para os outros servidores do grupo. Na implementação mais simples, o balanceador de carga detecta a integridade do servidor interceptando respostas de erro a solicitações regulares. As verificações de integridade do aplicativo são um método mais flexível e sofisticado no qual o balanceador de carga envia solicitações de verificação de integridade separadas e requer um tipo especificado de resposta para considerar o servidor íntegro.

NGINX: Balanceador de Carga

Outra função útil fornecida por alguns **balanceadores de carga** é a **persistência da sessão**, o que significa enviar todas as solicitações de um cliente específico para o mesmo servidor. Embora o HTTP seja **stateless** em teoria, muitos aplicativos devem armazenar informações de estado apenas para fornecer sua funcionalidade principal - pense no carrinho de compras em um site de comércio eletrônico. Esses aplicativos apresentam desempenho inferior ou podem até falhar em um ambiente com balanceamento de carga, se o balanceador de carga distribuir solicitações em uma sessão do usuário para servidores diferentes, em vez de direcioná-las para o servidor que respondeu à solicitação inicial.

NGINX: Proxy Reverso

Enquanto a implementação de um balanceador de carga só faz sentido quando você possui vários servidores, geralmente faz sentido implantar um proxy reverso com apenas um servidor Web ou servidor de aplicativos. Você pode pensar no proxy reverso como a "face pública" de um site. Seu endereço é o anunciado para o site e fica na borda da rede do site para aceitar solicitações de navegadores da web e aplicativos móveis para o conteúdo hospedado no site. local na rede Internet. Os benefícios são duplos:

Maior segurança - Nenhuma informação sobre seus servidores back-end é visível fora da rede interna, portanto, clientes mal-intencionados não podem acessá-los diretamente para explorar quaisquer vulnerabilidades. Muitos servidores proxy reversos incluem recursos que ajudam a proteger os servidores back-end contra ataques DDoS (distribuídos de negação de serviço), por exemplo, rejeitando o tráfego de endereços IP de clientes específicos (lista negra) ou limitando o número de conexões aceitas de cada cliente.

NGINX: Proxy Reverso

Maior escalabilidade e flexibilidade - como os clientes veem apenas o endereço IP do proxy reverso, você pode alterar a configuração da sua infraestrutura de back-end. Isso é particularmente útil em um ambiente com balanceamento de carga, onde você pode escalar o número de servidores para cima e para baixo para corresponder às flutuações no volume de tráfego.

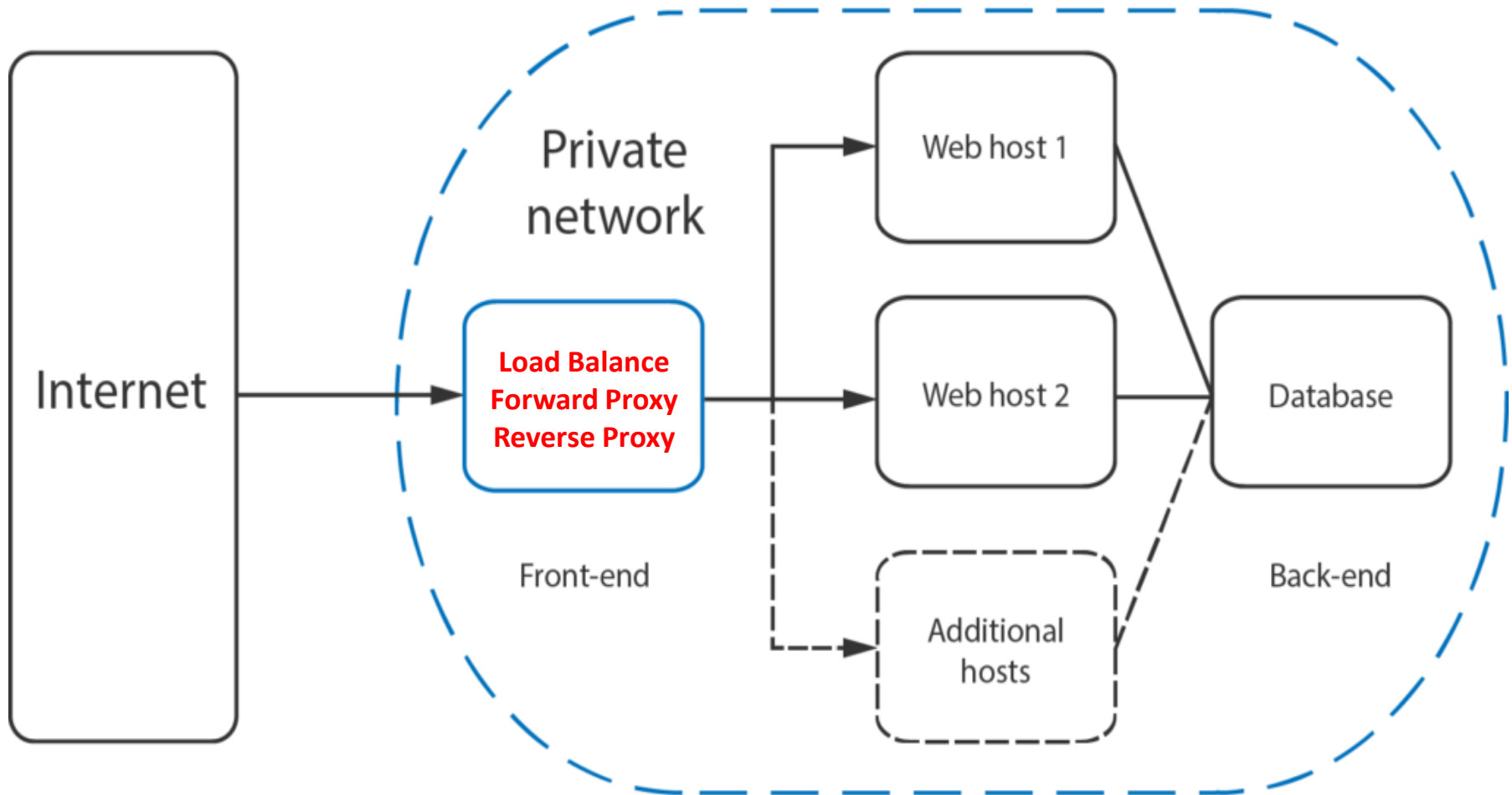
NGINX: Proxy Reverso

Outro motivo para implantar um proxy reverso é **WEB Acceleraion** - reduzindo o tempo necessário para gerar uma resposta e devolvê-la ao cliente. As técnicas para aceleração da web incluem o seguinte:

Compactação - A compactação das respostas do servidor antes de devolvê-las ao cliente (por exemplo, com gzip) reduz a quantidade de largura de banda necessária, o que acelera o trânsito pela rede.

Terminação SSL - Criptografar o tráfego entre clientes e servidores protege-o à medida que atravessa uma rede pública como a Internet. Mas a descriptografia e criptografia podem ser onerosas em termos computacionais. Ao descriptografar solicitações recebidas e criptografar as respostas do servidor, o proxy reverso libera recursos nos servidores back-end que eles podem dedicar ao seu objetivo principal, veiculando conteúdo.

Armazenamento em cache - Antes de retornar a resposta do servidor back-end ao cliente, o proxy reverso armazena uma cópia dele localmente. Quando o cliente (ou qualquer cliente) faz a mesma solicitação, o proxy reverso pode fornecer a resposta em si a partir do cache, em vez de encaminhar a solicitação ao servidor de back-end. Isso diminui o tempo de resposta do cliente e reduz a carga no servidor de back-end.



NODE.JS

- De modo simples é um server-side javascript engine, ou seja, executado no lado do servidor
- Construído sobre o engine V8 do Chrome
- Executado por linha de comando
- É um framework para aplicações em rede de alta performance, otimizado para ambientes de alta concorrência
- O .js do nome (node.js) não significa que é escrito totalmente em JavaScript. Consiste em 40% JS e 60% de C++
- Utiliza modelo de event-driven loop e non-blocking I/O o que o torna extremamente leve e rápido
- Totalmente single-thread; cpu-cycles intensive

NODE.JS – Non-Block I/O

- Esperar por pedidos de I/O degrada a performance
- O Node.JS utiliza eventos em JavaScript para anexar retornos às chamadas de I/O

Block I/O Call

```
...  
String sql = "SELECT c FROM Contato";  
  
Query q= em.createQuery(sql);  
  
List result = query.getResultList();  
...
```

Non-Block I/O Call

```
...  
var callback = function(err, rows) {  
    if(err) throw err;  
    console.log(rows);  
});  
  
con.query("SELECT * FROM Contato",  
callback);  
...
```

NODE.JS – Non-Block I/O

- I/O Tradicional

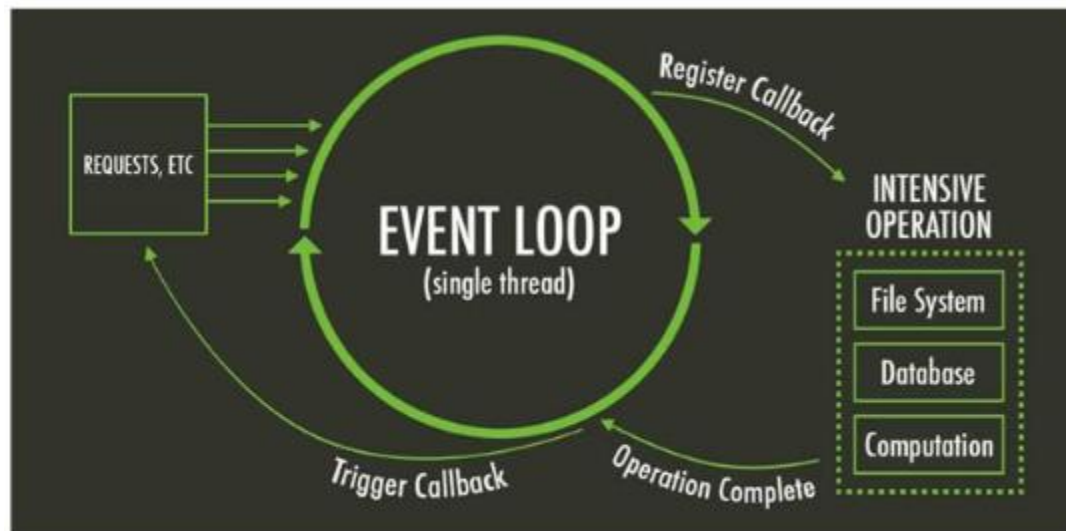
```
var result = db.query("select x from table_Y");  
doSomethingWith(result); //wait for result!  
doSomethingWithoutResult(); //execution is blocked!
```

- I/O Não-Tradicional - Não-Blocante

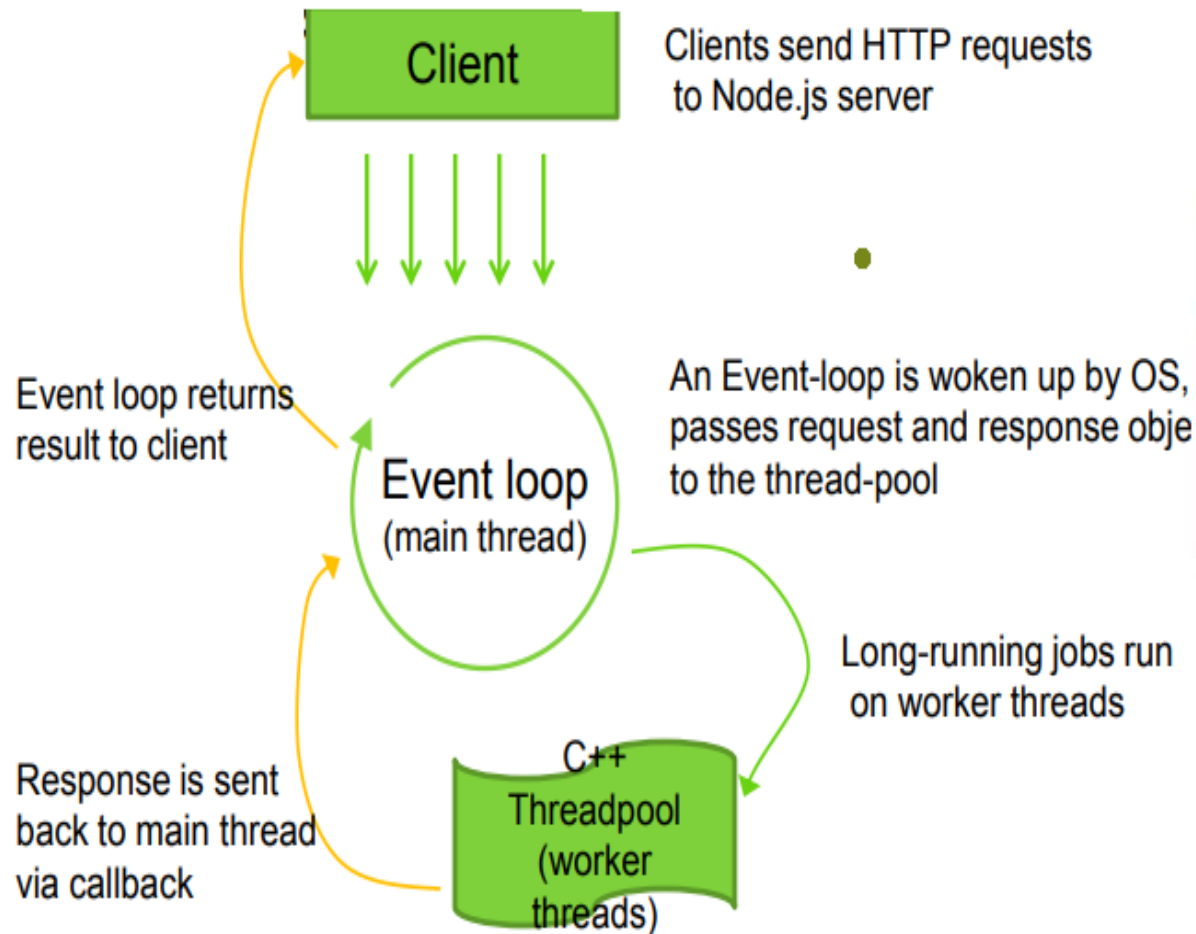
```
db.query("select x from table_Y",function (result){  
    doSomethingWith(result); //wait for result!  
});  
doSomethingWithoutResult(); //executes without any  
    delay!
```


NODE.JS – Event-Driven Loop

- Ao invés de threads, o Node.JS utiliza um controle por loop de eventos aliviando assim o overhead da troca de contexto
- Event Loops são o core de programação event-driven, sendo que todas as interfaces de programas atuais utilizam event-loops para rastrear um determinado evento



NODE.JS – Event-Driven Loop



JavaScript		C/C++		
node standard library				
node bindings (socket, http, etc)				
V8	thread pool (libeio)	event loop (libev)	DNS (c-ares)	crypto (OpenSSL)

O que é possível fazer com Node.JS ?

- Criar um servidor HTTP e mostrar a palavra “hello-word” num navegador com apenas 4 linhas de código JavaScript
- Criar um servidor TCP similar a um servidor HTTP em apenas 4 linhas de código JavaScript
- Criar um servidor DNS
- Criar um servidor de arquivos (File Server) estático
- Criar uma aplicação do tipo Web Chat (GTalk) num navegador
- Também pode ser utilizado para criação de jogos on-line; ferramentas de colaboração; serviços de streaming; ou qualquer coisa que tenha de enviar atualizações para usuário em tempo real

Node.JS Ecosystem

- Node.js heavily relies on **modules**, in previous examples **require** keyword loaded the http & net modules.
- Creating a module is easy, just put your JavaScript code in a separate js file and include it in your code by using keyword require, like:

```
var modulex = require('./modulex');
```

- Libraries in Node.js are called packages and they can be installed by typing

```
npm install "package_name"; //package should be  
available in npm registry @ nmpjs.org
```

- **NPM** (Node Package Manager) comes bundled with Node.js installation.

Node.JS Ecosystem: Connect DB (MongoDB)

- Install mongojs using npm, a MongoDB driver for Node.js

```
npm install mongojs
```

- Code to retrieve all the documents from a collection:

```
var db = require("mongojs")
    .connect("localhost:27017/test", ['test']);
db.test.find({}, function(err, posts) {
    if( err || !posts) console.log("No posts found");
    else posts.forEach( function(post) {
        console.log(post);
    });
});
```

Node.JS Ecosystem: Twitter Streaming

- Install nTwitter module using npm:

```
Npm install ntwitter
```

- Code:

```
var twitter = require('ntwitter');  
var twit = new twitter({  
  consumer_key: 'c_key',  
  consumer_secret: 'c_secret',  
  access_token_key: 'token_key',  
  access_token_secret: 'token_secret'});  
twit.stream('statuses/sample', function(stream) {  
  stream.on('data', function (data) {  
    console.log(data);  
  });  
});
```