Resource Containers

A new facility for resource management in server systems

G. Banga, P. Druschel, J. C. Mogul

OSDI 1999

Presented by Uday Ananth

Lessons in history..

Web servers have become predominantly responsible for a users' **perceived** computing performance

These servers must often **scale** to millions of clients.

A lot of work has been done for improving the performance of web servers and making them more scalable.

Service providers want to **exert explicit control** over resource consumption policy. (Differentiated QoS)

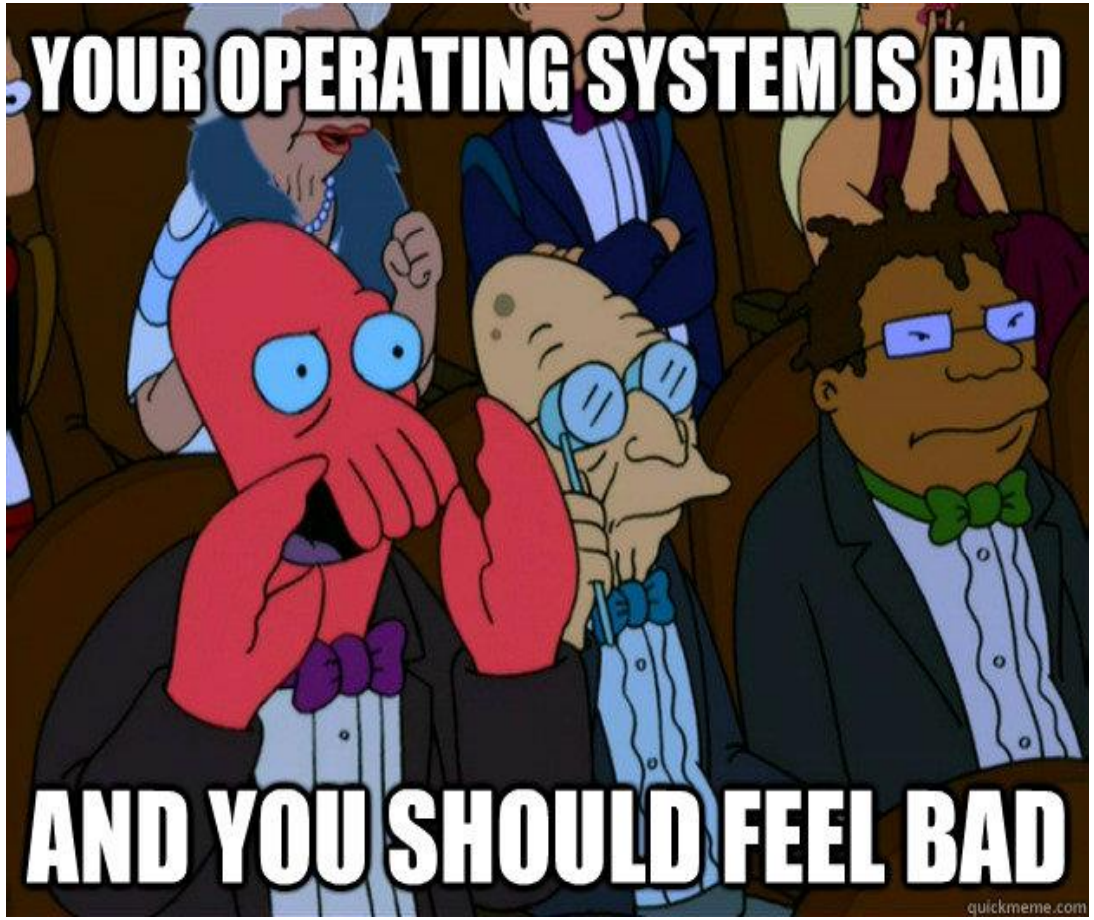
Clip

The blame game..

There are **shortcomings in the resource management** abstractions.

Operating systems treat processes as the unit of resource management.

Web servers use **a small set of processes** to handle several activities, making them **too coarse to be the right unit** of resource management.

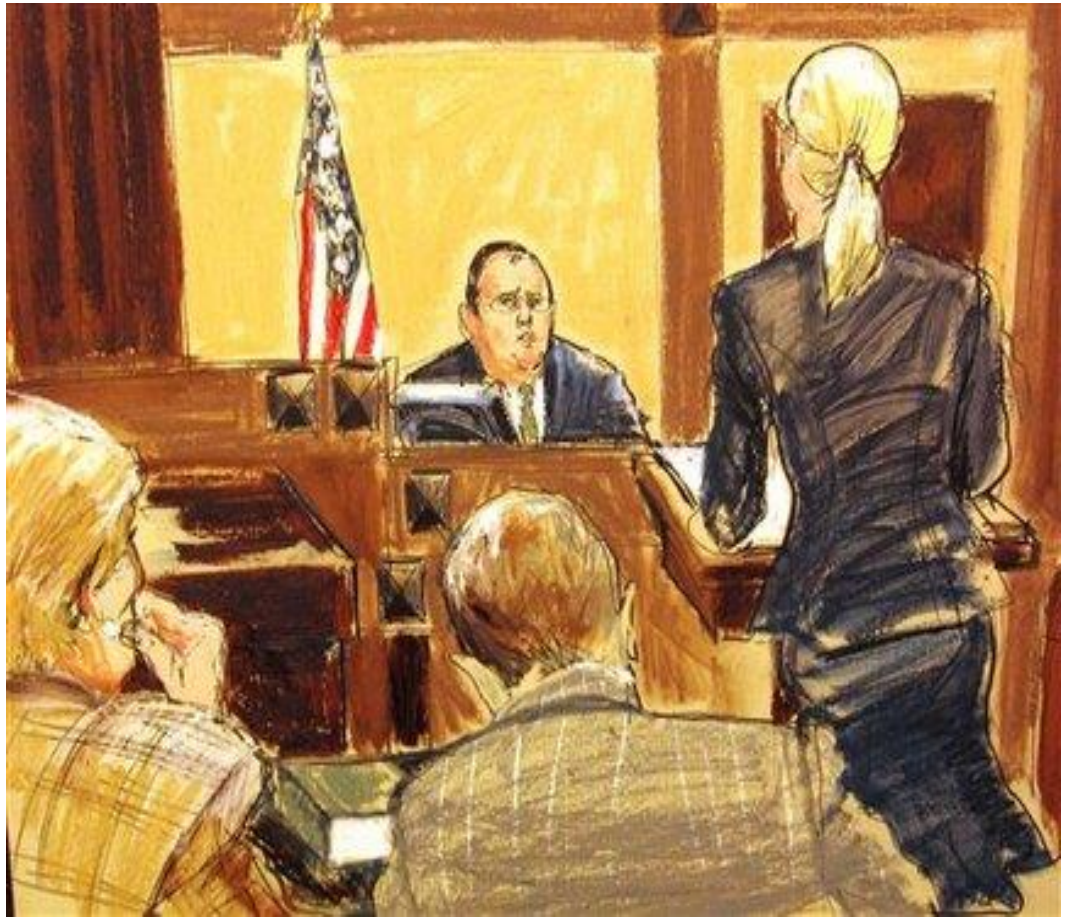


Let’s draw up the terms..

Resource Principals are entities for which separate resource allocation and accounting are done.

Protection domains are entities that need to be isolated from each other**.**

In most operating systems, **processes** or **threads** are both resource principals as well as protection domains.

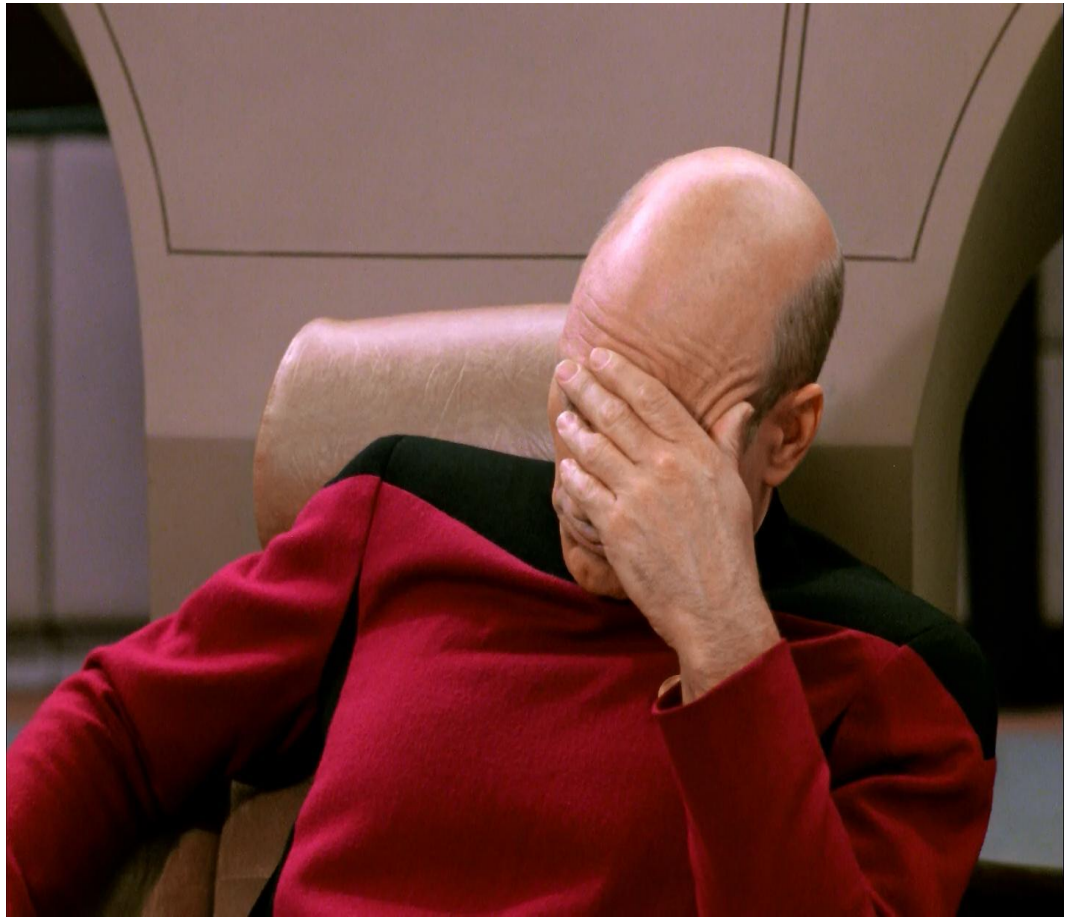


The problems..

Protection domain and Resource principal exist in the **same process abstraction**.

Applications have **little control** over the resources the kernel consumes for them.

The resources utilized by the kernel are often **accounted / utilized inaccurately** (according to the process) resulting in **bad scheduling decisions**.

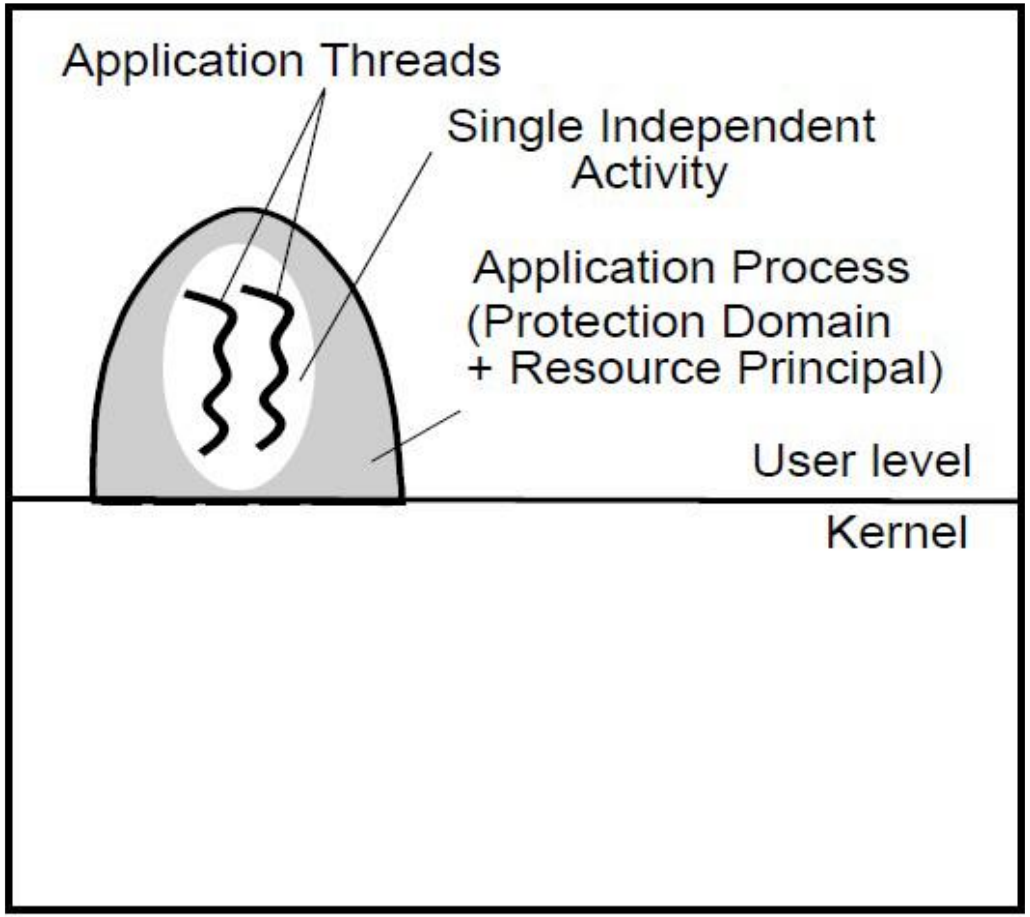


The assumptions we make.. (not necessarily wrong..)

Most user space applications are a single process, (possibly consisting of **multiple threads**) and perform a single activity.

Resources consumed by the process are properly accounted, as the kernel consumes few resources on behalf of the application.

Therefore, the **process is an appropriate resource principal**.



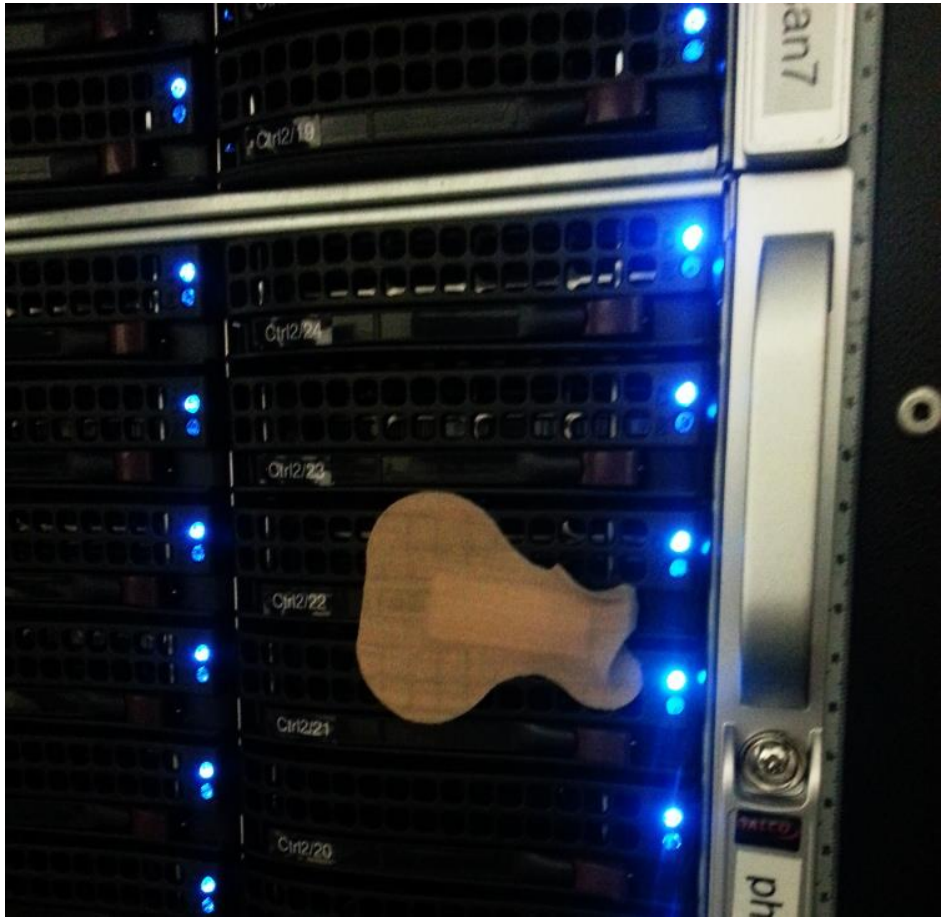
The Band-Aids we applied..

Let's take a look at the previous approaches we've tried.

Process-per Connection

Single-Process Event-Driven Connection

Multi-threaded Server



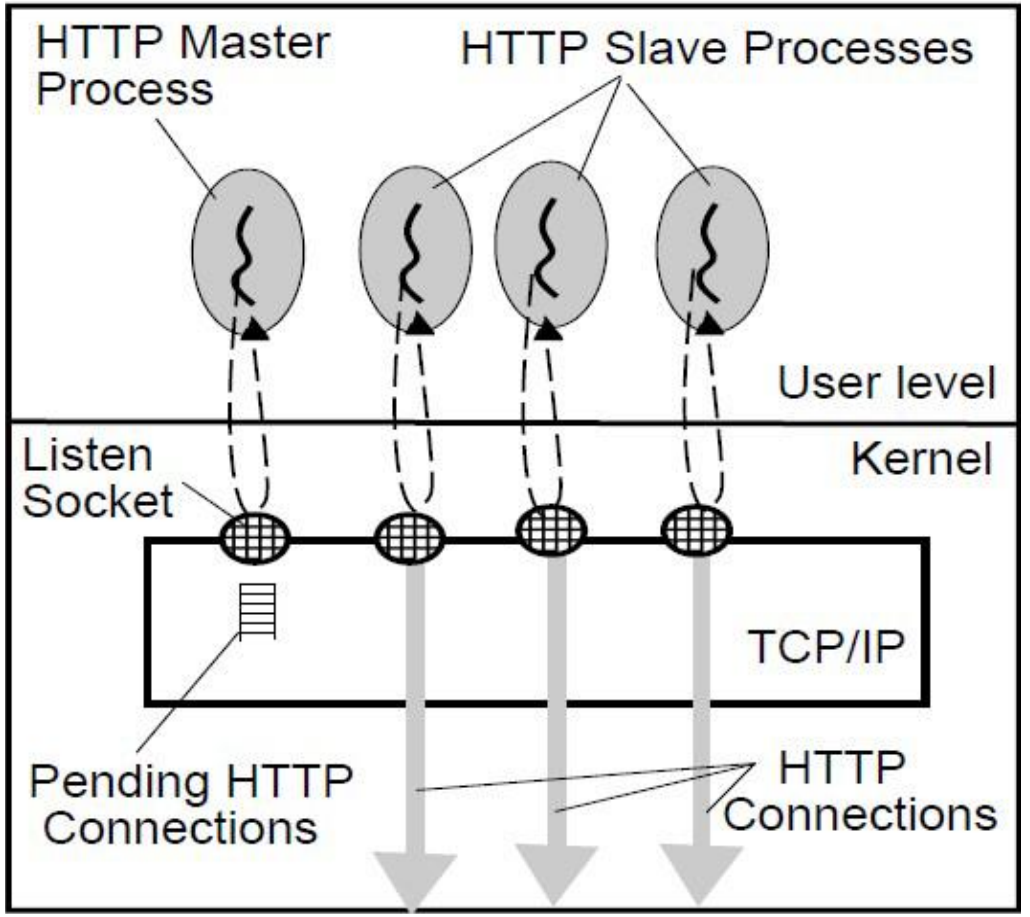
Process-per Connection..

Simple to implement.

We have a **master process listening to a port** for new connection requests.

For each new connection a **new process is forked**.

There are caveats such as **Forking overhead, context switching overheads, IPC overheads**.



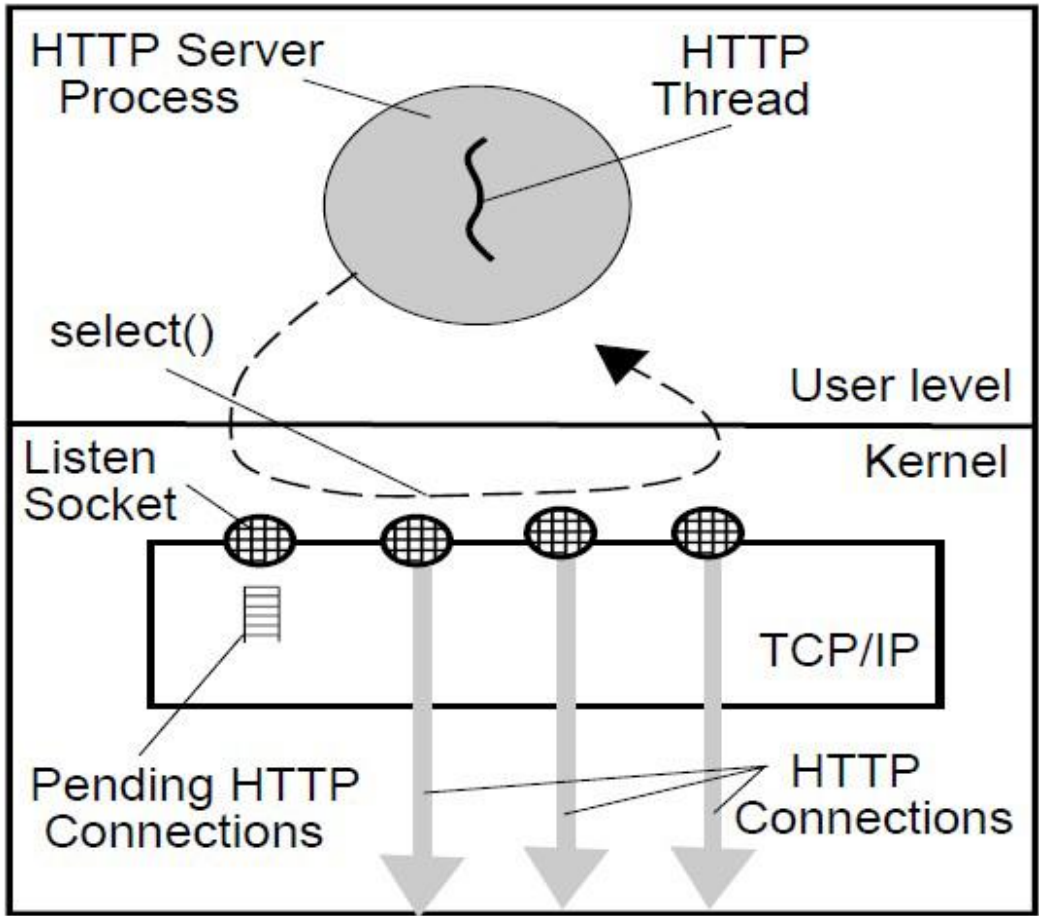
Single-Process Event-Driven Connection..

Harder to implement.

Single process **runs event handlers** in the main loop for each ready connection in the queue.

Helped **avoid IPC and context switches** and hence scaled better.

They weren't really **concurrent** unless they ran **on multi-processor systems**. (They could fork into multiple processes)



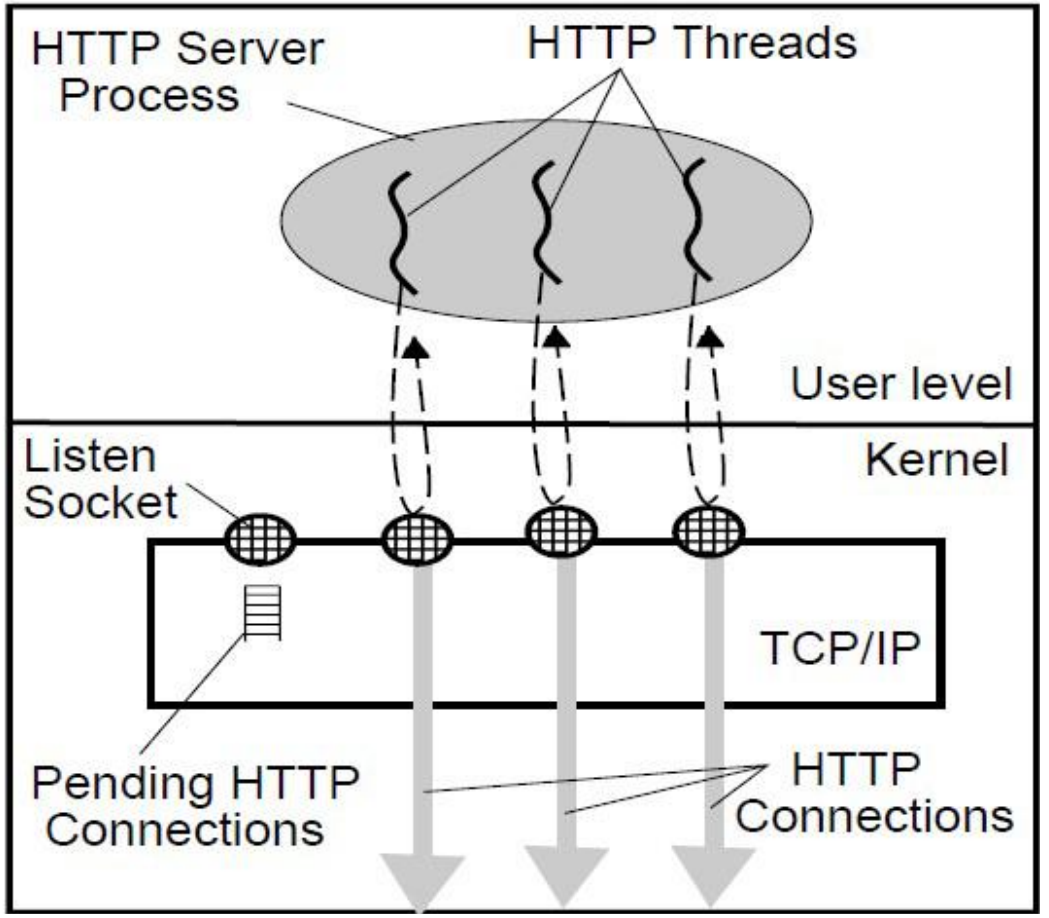
Multi-threaded Server..

Easier to implement compared to event driven models.

Created threads for each incoming connection or created **multiple threads** and Idle threads listened to incoming connections.

Threads are **scheduled by thread scheduler**.

Avoided context switches and **scaled better**.

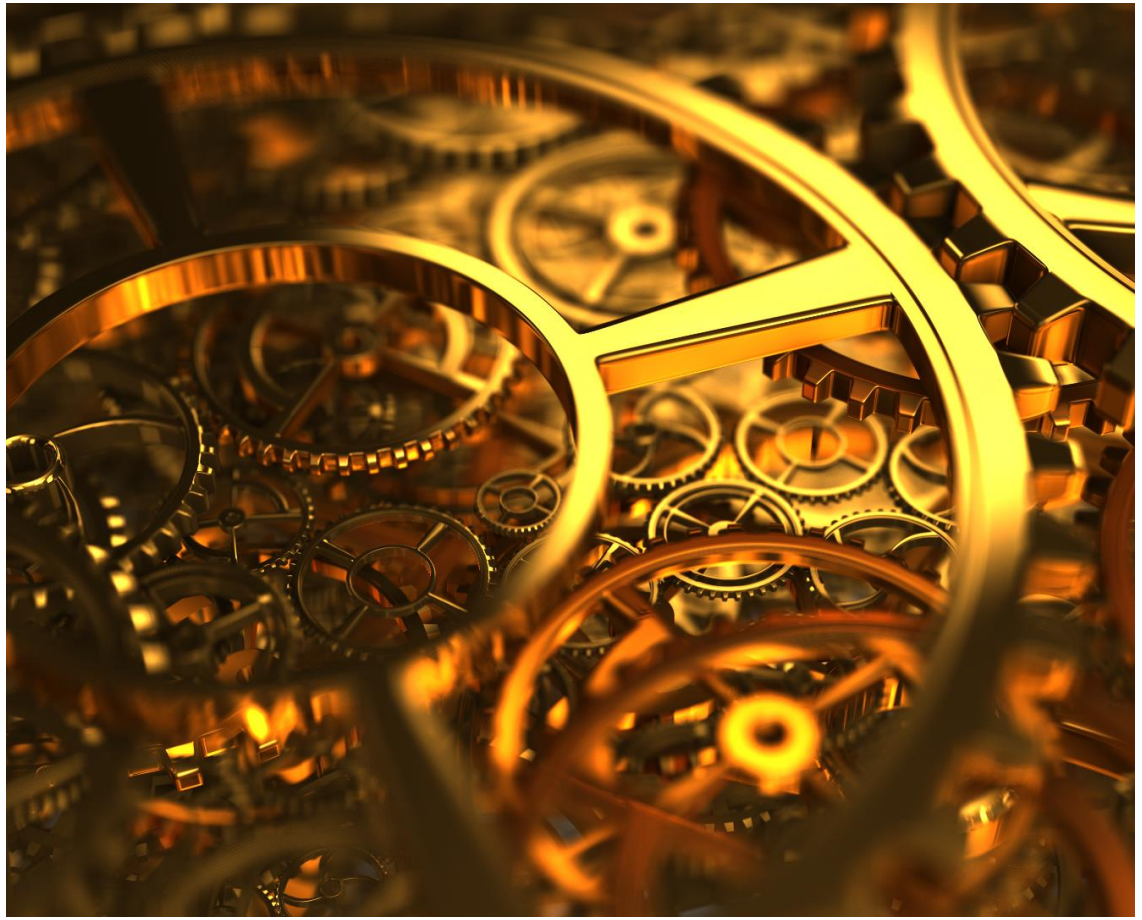


What's happening behind the scenes..

Dynamic pages **required multiple resources** and were created in response to user input.

Multiple processes may have been created to handle the dynamic request and required some **overhead of IPC**.

Kernel handles network processing for buffers, sockets, etc. Those operations are **separate from server app** and charged to either one or any unlucky process!



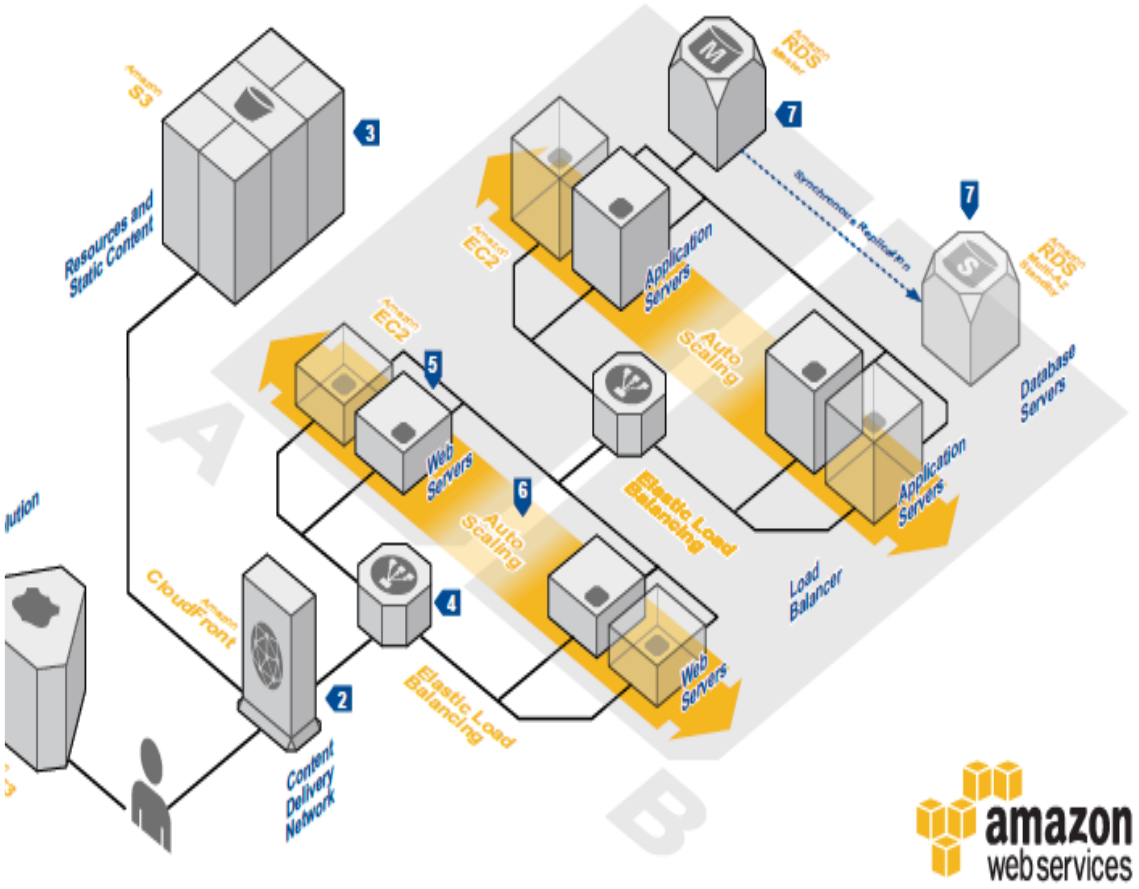
Survival of the fittest..

Let's have a look at how we've managed to evolve over the application space and where we fail by using a separate domains.

A network-intensive application

A multi-process application

Single-process multi-threaded application

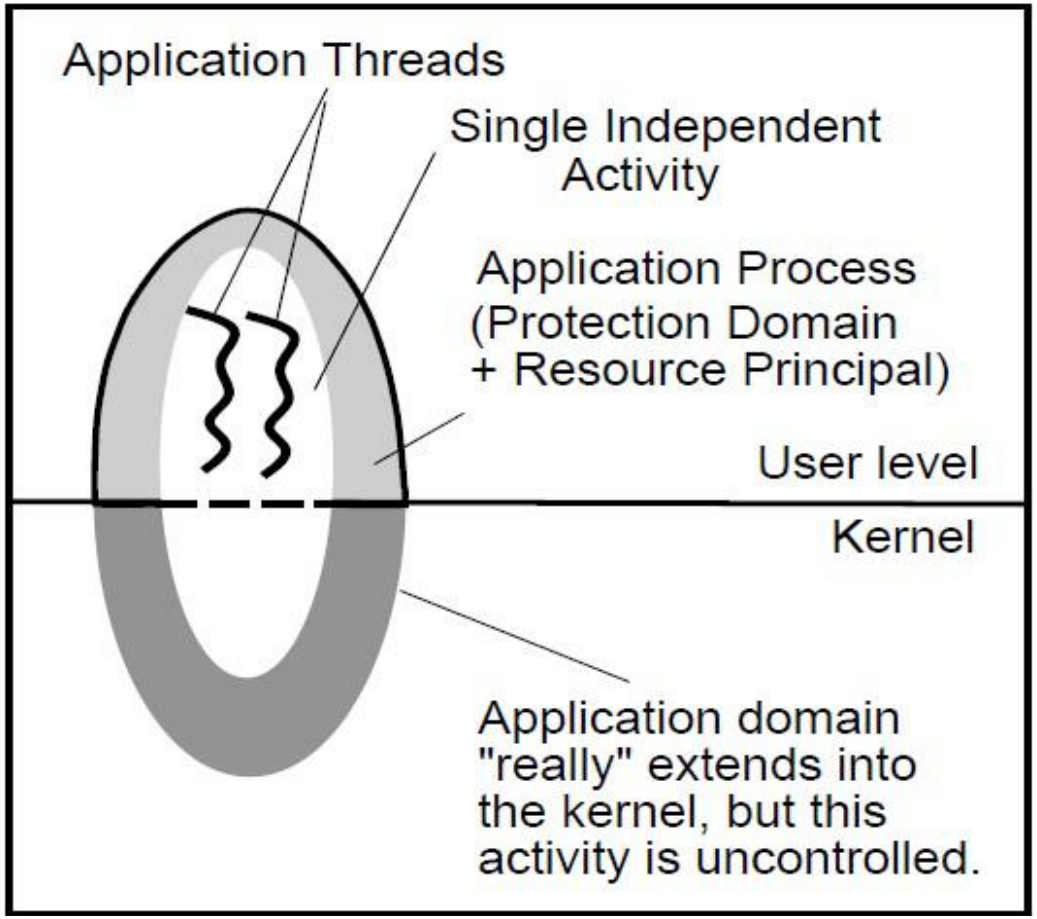


A network-intensive application..

A process consisting of **multiple threads**, performing a **single activity**. The process is the right unit for protection, but it does not encompass all the resources being consumed.

Referred to as **eager processing** and results in **inaccurate accounting**.

We are unable to charge an application for the processing that the kernel does.

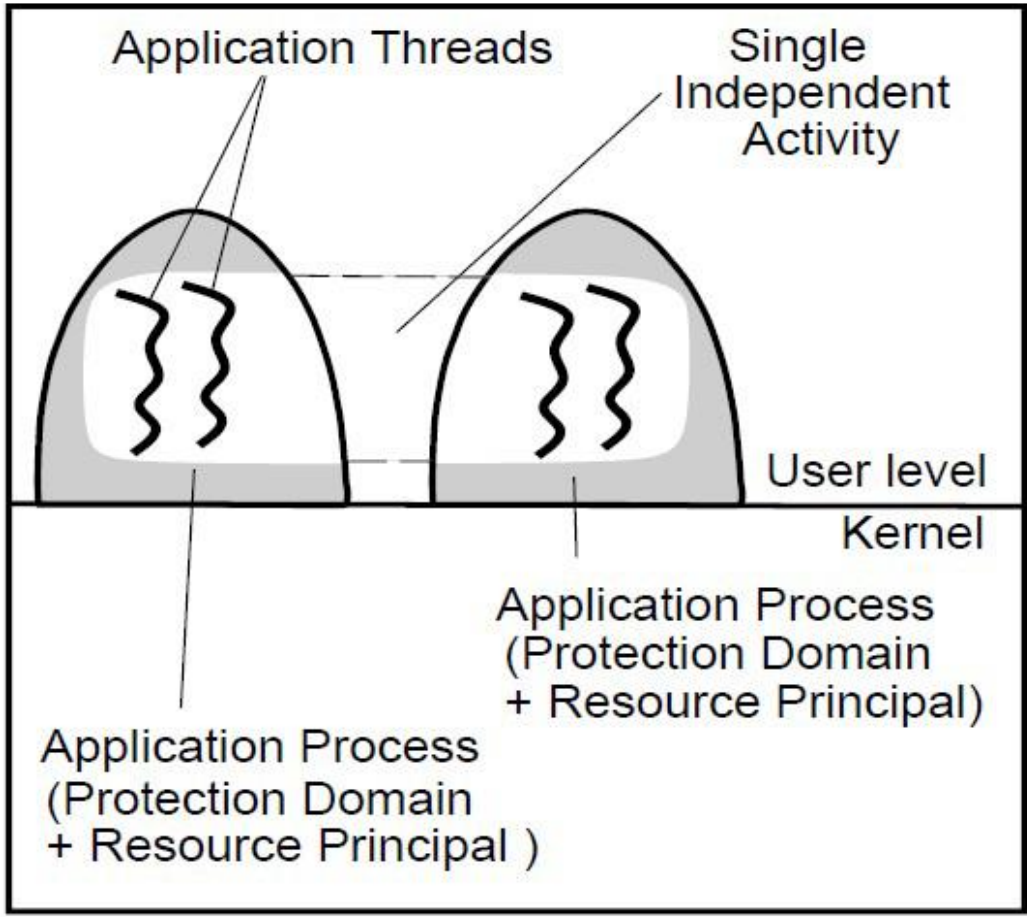


A multi-process application..

Composed of **multiple user space processes**, cooperating, to perform a single activity.

Resource management is a **set of all the processes** rather than of individual processes.

Eg. Parallel Simulation

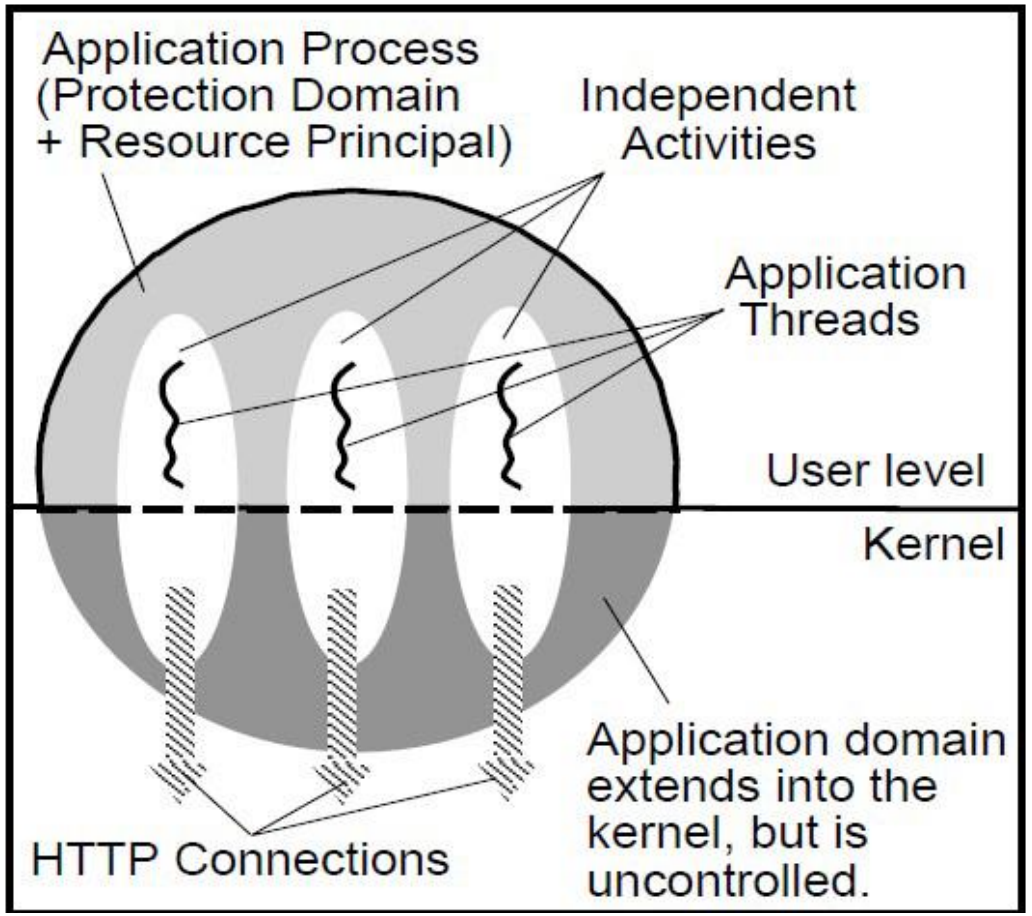


Single-process multi-threaded application..

There is **a single process using multiple independent threads**, one for each connection.

The correct unit of **resource management is smaller** than a process.

Resource management is the set of all resources used to accomplish a single independent activity.



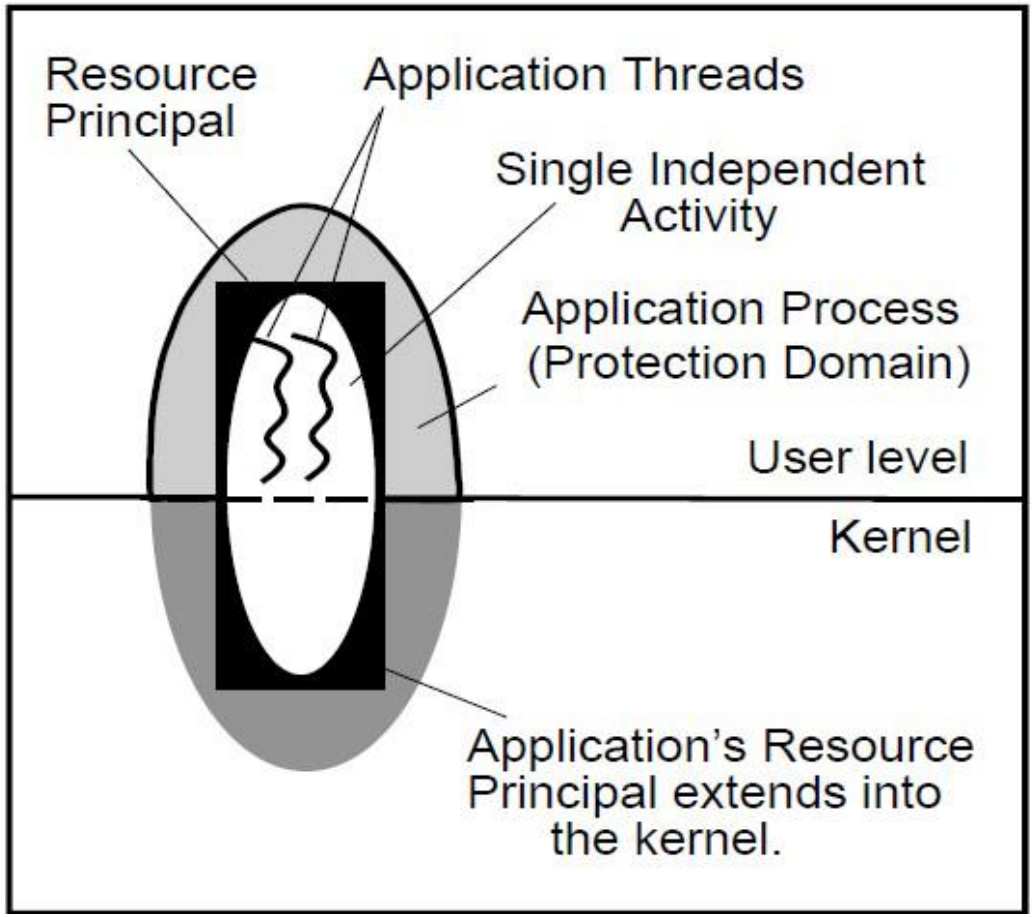
Lazy Receiver Processing..

Integrates network processing and resource management.

When a packet arrives, instead of doing all the protocol processing, **LRP** does some **minimal processing**.

The remainder will be performed by the process for which the packet was intended.

Brings **equivalence** between resource principal process.

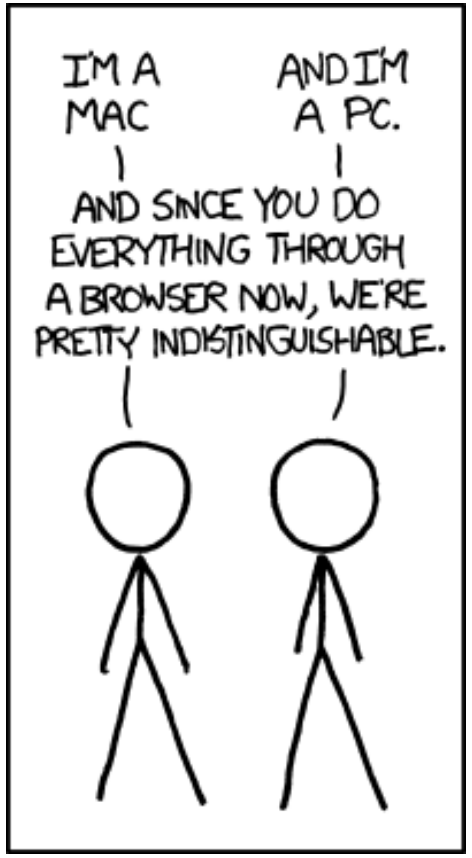


A quick recap..

Dual functions of a protection domain and resource principal are **not efficient.**

The system **does not** allow applications to **directly control** resource consumptions (e.g. priority) or management.

There may be a requirement from Web servers to provide some kind of **guarantee to clients** (differentiated QoS), making accurate accounting necessary.



A solution..

Containers are an **abstract entity** that logically contain all system resources being used by an app to achieve a particular independent activity.

Containers can contain **resources** like CPU time , Sockets, Control Blocks, Network buffers, etc.

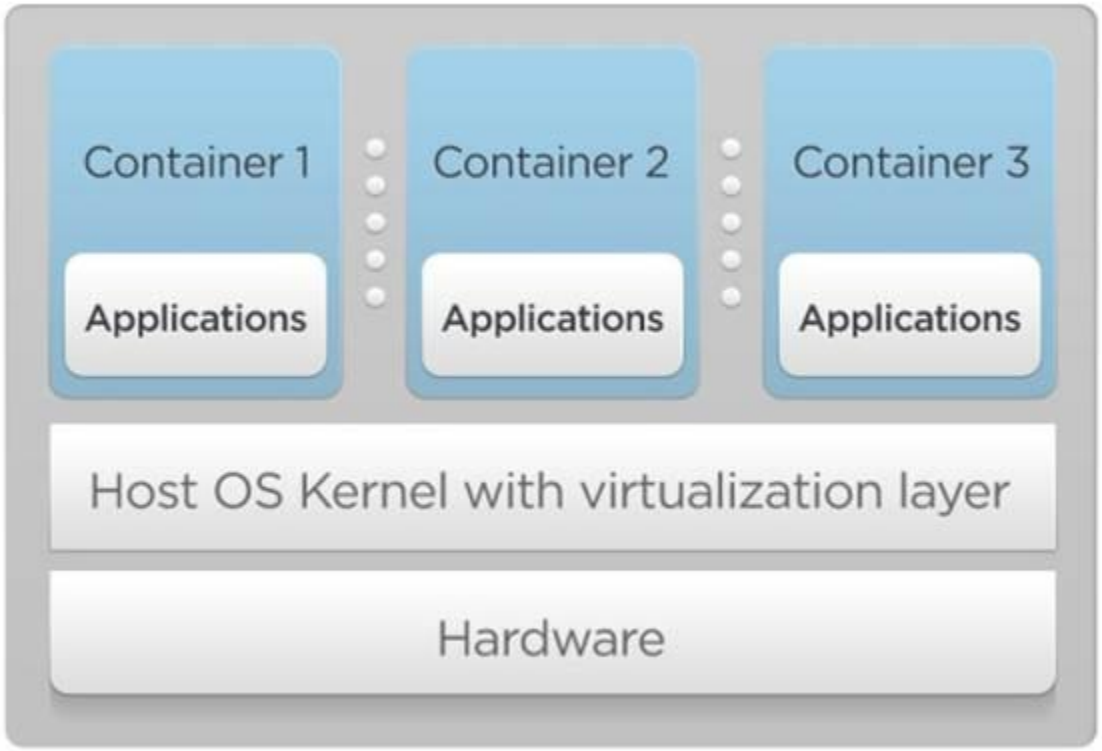
Containers can also be attached with **attributes** to limit resources such as CPU availability, network QoS, scheduling priorities, etc.

How does it work?

Applications have to **identify resource principals** and associate those independent activities with resource containers.

Resource binding is the **relation between** resource / processing **domains** and the associated resource **principals**, thereby, allowing it to charge for resources within kernel.

Dynamic resource binding is based on the **activity or purpose** that a thread / process is serving. This allows a thread to be associated with multiple resource containers.

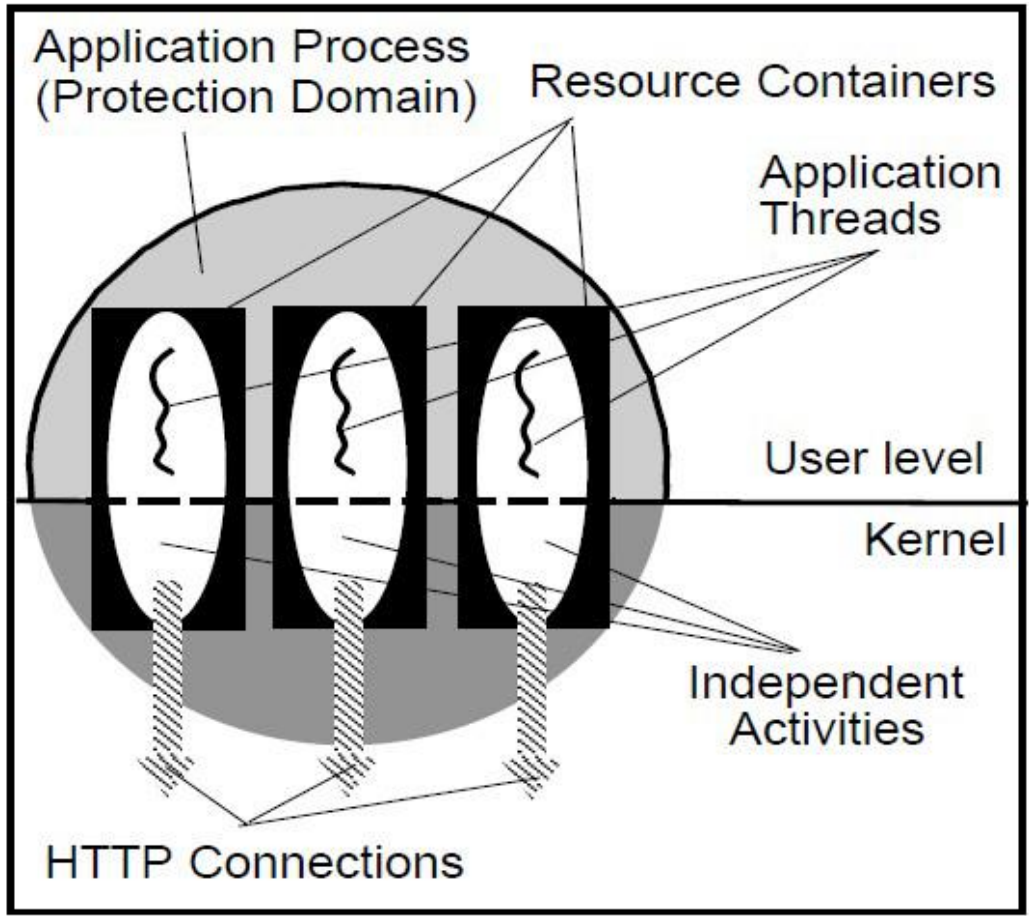


Containers in a multi-threaded server..

The server creates **a new container for each connection** handled by a single **thread bound** to the container.

Kernel processing is charged to this container.

The scheduling priority of the associated thread would **decay if the thread consumes more** than it's fair share of resources.

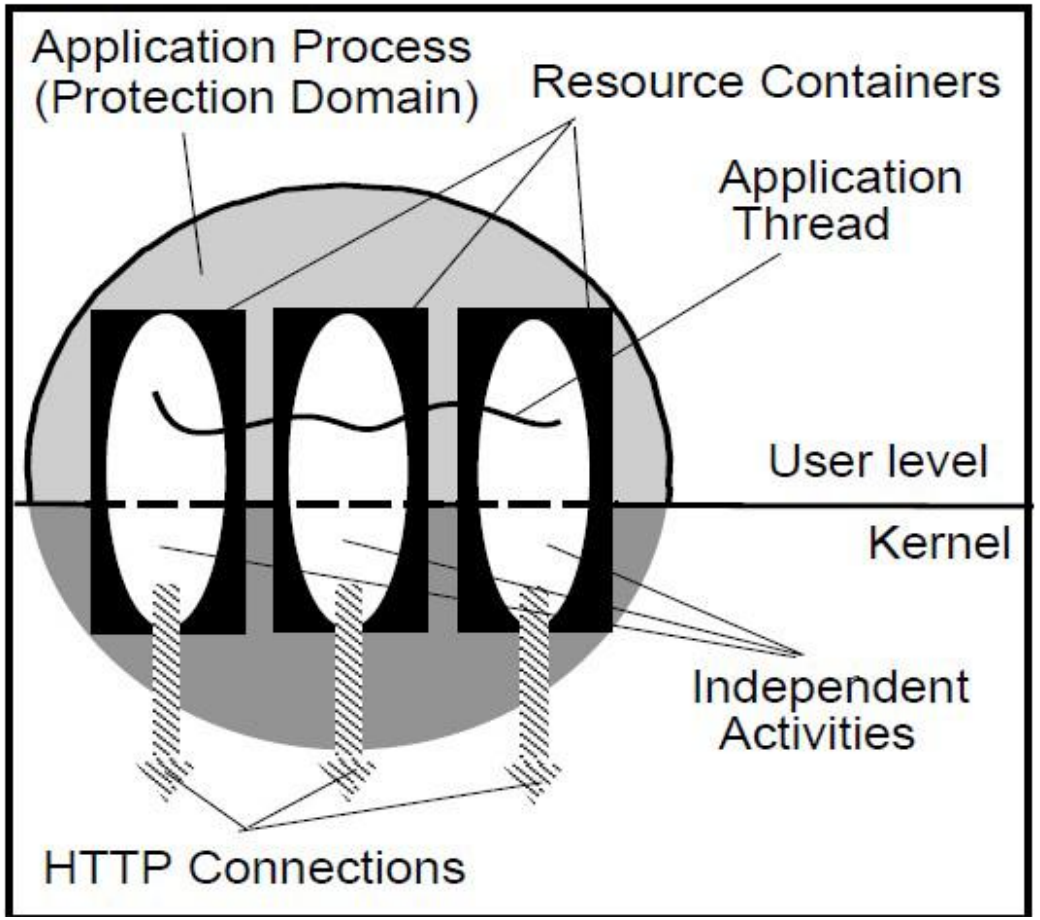


Containers in an event driver server..

The web server would associate a **new container for each connection**. However, they would all be **serviced by a single thread**.

The thread's **binding would be changed dynamically** as it moves across the connections.

The associated **container will be charged for the processing** the thread performs for them.



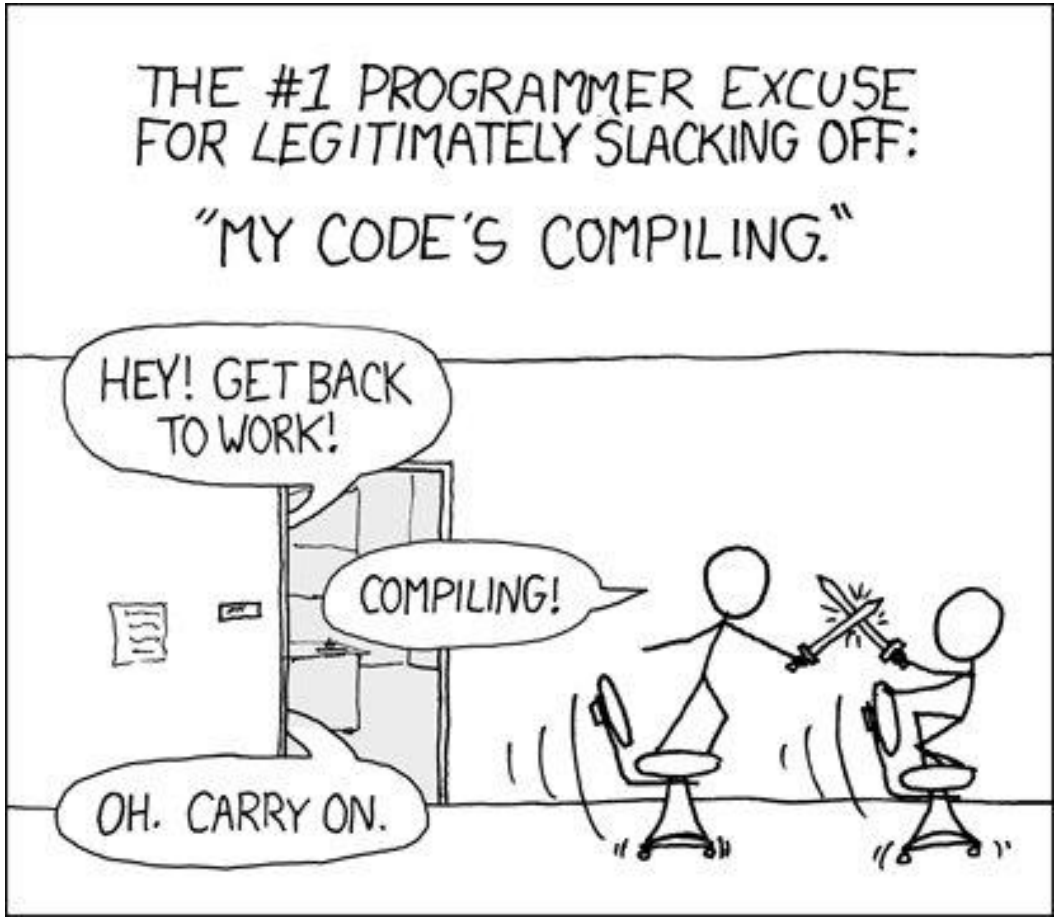
How do you build it?

There were several modifications to Digital UNIX 4.0D kernel.

CPU scheduler was modified to treat containers as resource principals that could obtain a **fixed share** of time, or **share resources** assigned to its parent along with its siblings.

The TCP/IP subsystem was modified to implement Lazy Receiver Processing.

The Server software was a single-process, **event-driven** derived from **thttpd** and the Clients used the S-Client software.



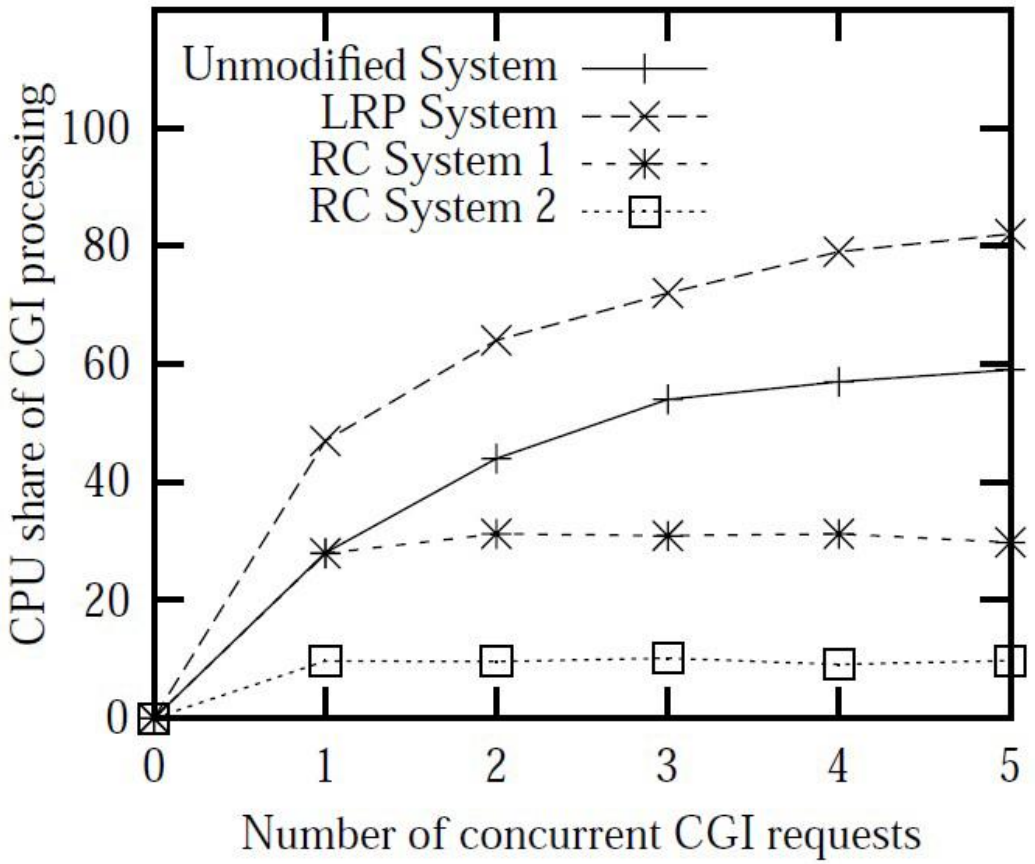
Can it control resource utilization?

Demonstration on how resource can **isolate static requests** being serviced, from **excessive Dynamic CGI requests**.

CGI requests consumed about **2 seconds** of CPU time.

Static requests were serving a **1 KB document** in the cache.

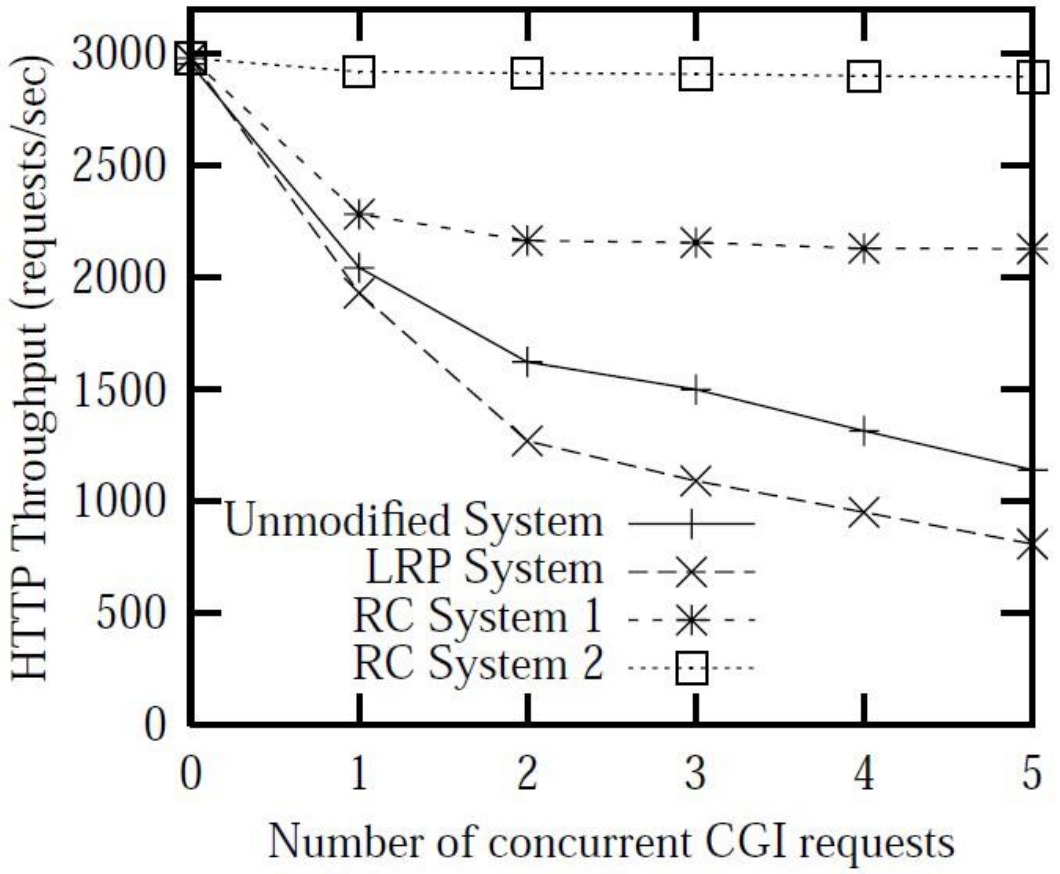
The CGI processes were restricted to **30% (RC 1) and 10% (RC 2)**.



What about performance?

An extension of the resource constraints was to **measure throughput** under the pressure of many dynamic requests.

Restricting resource usage of dynamic requests allows the OS to deliver a certain degree of **fairness** across containers.



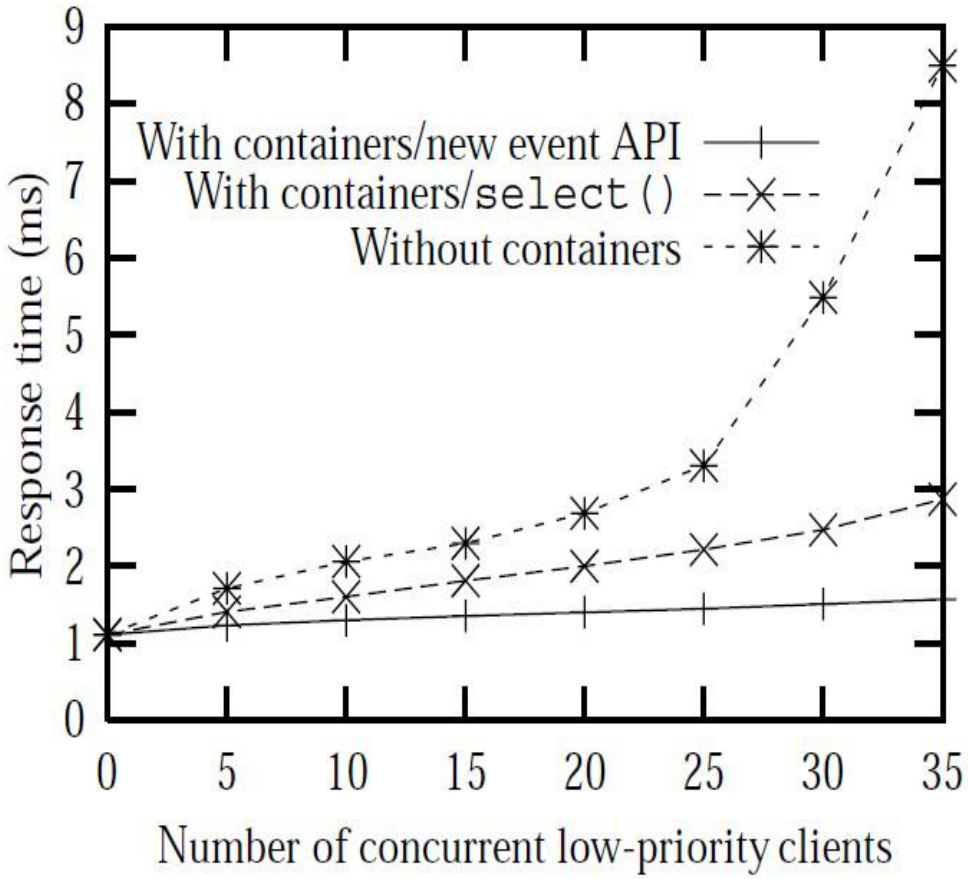
How about QoS guarantees?

An experiment tested the effectiveness of resource containers in the **prioritized handling** of clients

A high priority client should see the **guaranteed QoS** even as the load imposed by the low priority clients is increased.

A response time of **1 msec** for the high priority client is chosen.

Works, and a slight degradation is seen when select( ) is used (because of scalability problems in select).



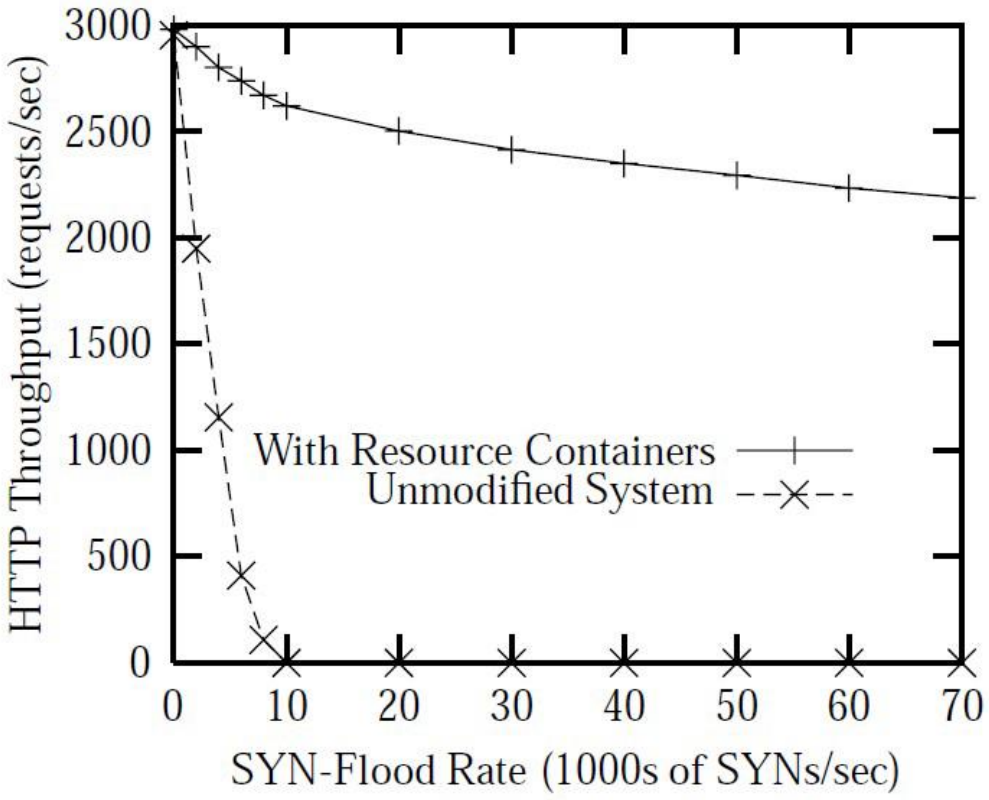
Can it overcome the dark side?

Containers help provide a certain amount of **immunity** against a DOS attack by SYN flooding.

A set of malicious clients sent bogus SYN packets to the server's HTTP port.

The **concurrent throughput** to other clients is **measured**.

The **slight degradation** is due to the interrupt processing of the SYN flood.



So..

Containers can be used as an **abstraction** to explicitly **identify a resource principal**.

Containers **decouple resource principals from protection domains**.

Containers **allow explicit and fine-grained control** over resource consumption at both user-level and kernel-level.

They can provide **accurate resource accounting** enabling web servers to provide **differentiated QoS**.

What's the impact?

They are **not heavy** and **more efficiently** than hypervisors as Containers are based on shared OS resources.

There is considerable implications for **application density** as well tuned container systems, can see **four-to-six** times as many server instances compared to traditional hypervisors.

Can make a huge impacts for enterprise data centers and application developers.

Q&A..

I'm tired.. Leave me alone.. Please..

You may provide feedback via uday@vt.edu..