CONCURRENT

DISTRIBUTED TASK SYSTEM IN PYTHON

Created by Moritz Wundke

INTRODUCTION

Concurrent aims to be a different type of task distribution system compared to what MPI like system do. It adds a simple but powerful application abstraction layer to distribute the logic of an entire application onto a swarm of clusters holding similarities with volunteer computing systems.

INTRODUCTION CONT'D

Traditional task distributed systems will just perform simple tasks onto the distributed system and wait for results. Concurrent goes one step further by letting the tasks and the application decide what to do. The programming paradigm is then totally async without any waits for results and based on notifications once a computation has been performed.

WHY?

- Flexible
- Stable
- Rapid development
- Python

WHY? - FLEXIBLE

- Keystone for every framework
- Must be reused in many typed of projects
- Easy to integrate in existing code

WHY? - STABLE

- Fault tolerance
- Consistency
- We scarify availability

WHY? - RAPID DEVELOPMENT

- Fast prototyping
- Python is fast to write, very fast!
- Deliver soon and often principle

WHY? - PYTHON

- Huge community
- Used in scientific computation
- Deliver soon and often principle
- Imperative and functional
- Very organized
- Cross-Platform
- Object serialization through pickle, thus dangerous if not used properly!
- Drawback: pure python performance

THE DARK SIDE

- Pure python performance
  - Cython
  - Native
- No real multithread
  - The GIL issue
  - Releasing the GIL manually
  - multiprocessing (fork)

THE DARK SIDE - PERF

- Python is slow, thats a fact
- But we can boost it using natives
  - **Cython:** Static C compiler combining both python flexibility and C performance.
  - **Native c modules:** Create python modules directly in C.

THE DARK SIDE - GIL

- **Global Interpreter Lock:** Only one line or python object accessed at a time per process.
- We can release the GIL using natives like Cython or directly in a native module.
- We can also use processes instead of threads, while adding the need for IPC mechanisms.
- Shared vs Distributed memory / Threads vs Processes.

CONCURRENT

- Distributed task execution framework which tries to solve the GIL issue.
- Does not use threads when executing processes.
- Features different ways to implement IPC calls.
- All nodes in the system communicate through RPC calls or using HTTP or low-level TCP.
- Integrated Cython for performance tweaking.

OTHER FRAME WORKS

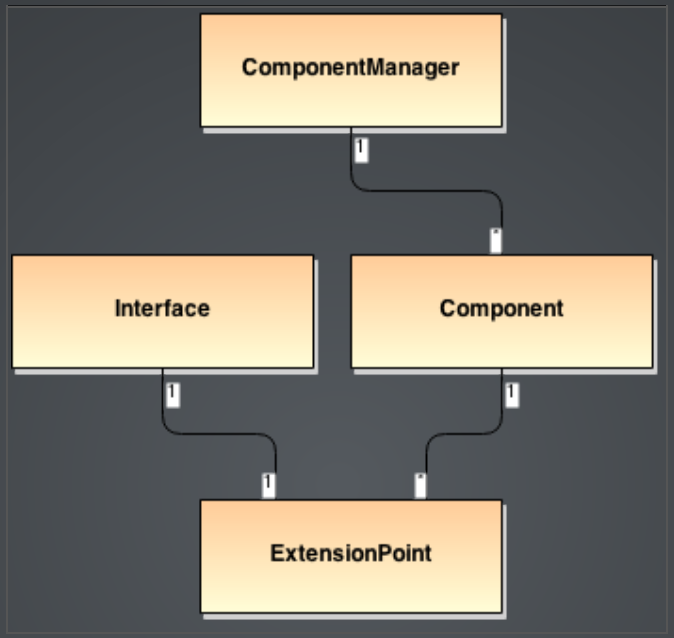
- **Dispy:** Fork based system, not applciation or cloud oriented as concurrent, problems with TCP congestion.
- **ParallelPython:** Thread based based system, not applciation or cloud oriented as concurrent, problems with TCP congestion.
- **Superpy:** Similar to concurrent but does not feature a high-performance transport layer. Only for windows.
- More libraries

ARCHITECTURE

Concurrent is build upon a flexible plug-able component framework. Most of the framework is plug-able in few steps and can be tweaked installing plug-ins.

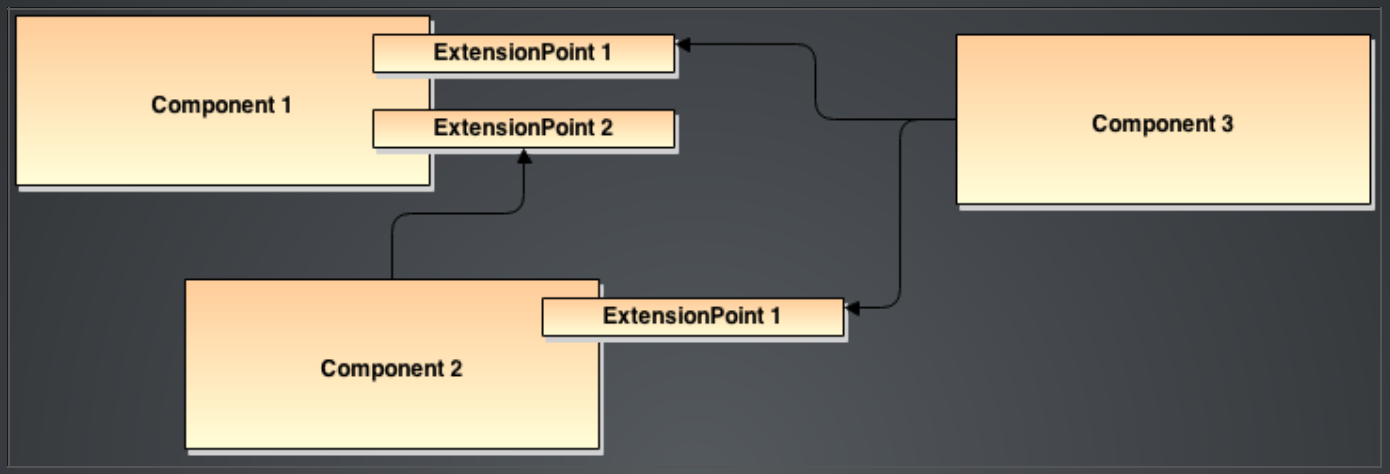
Applications themselves are plug-ins that are then load on the environment and executed.

COMPONENTS



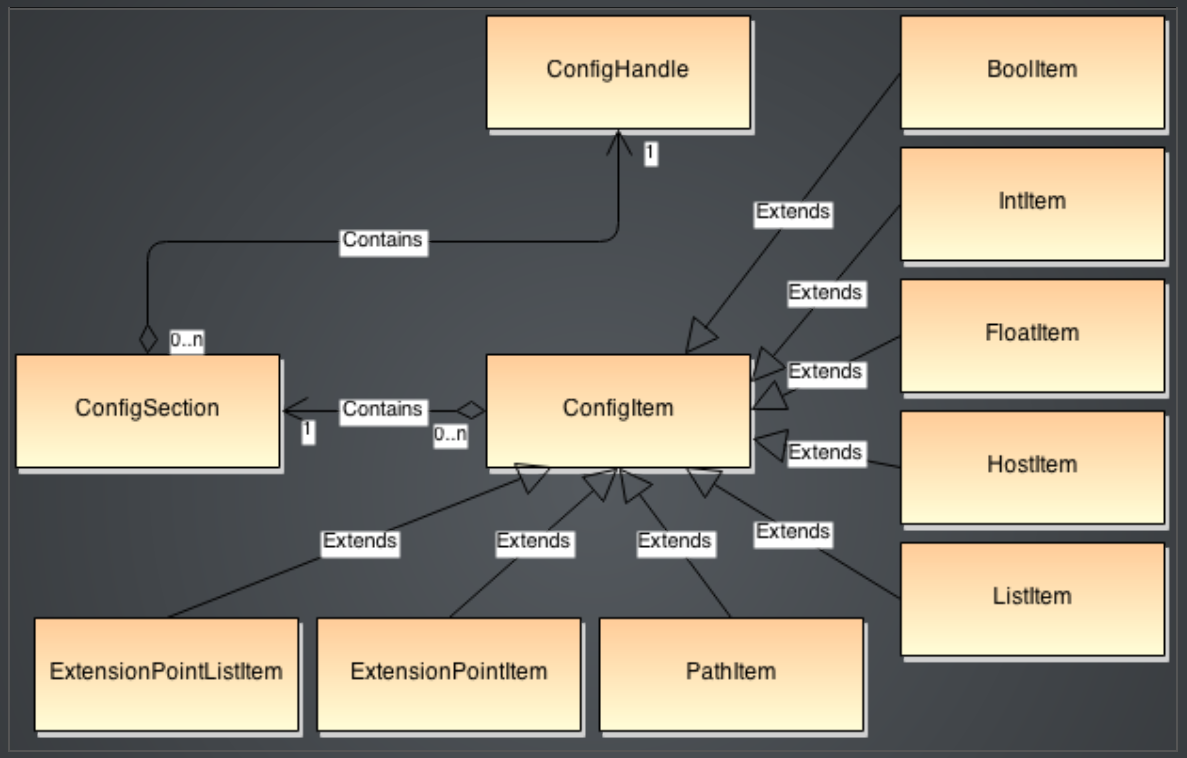
Components are singleton instances per *ComponentManager*. They implement the functionality of a given *Interface* and talk to each other through an *ExtensionPoint*.

COMPONENTS CONT'D



Example setting of *Components* linked together with *ExtensionPoints*

COMPONENTS CONT'D



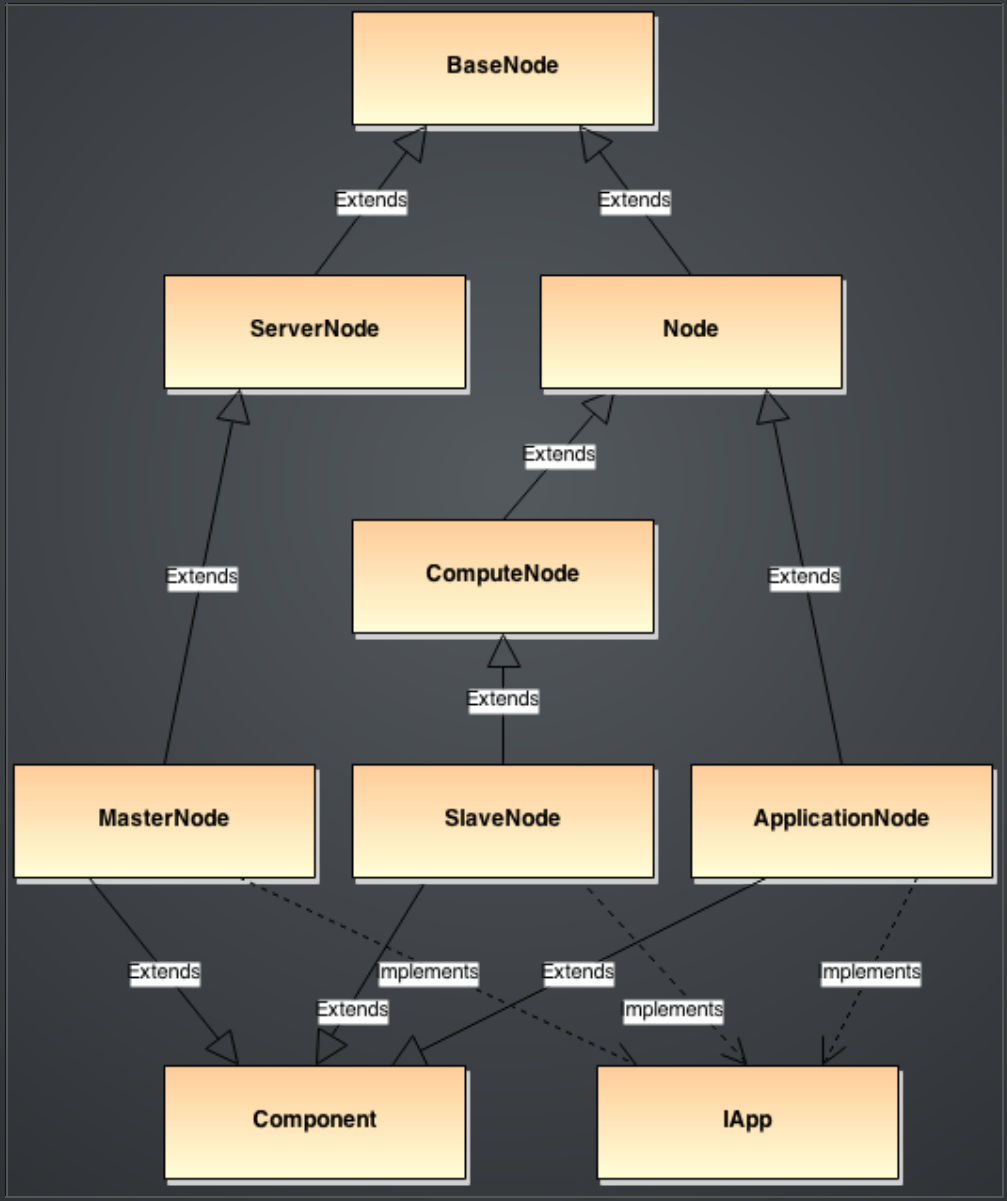
Configuration system allows us to configure *ExtensionPoints* via config files.

NODES

Our distributed system is based on a 3 node setup, while more classes are involved for flexibility

- **MasterNode:** Our main master (or a layer within a multi-master setup). Distributed the workload and maintains the distributed system.
- **SlaveNode:** A slave node is connected to a master (or a set of masters in a multi-master setup). Executes the workload a master sends to this node or requests work from it.
- **ApplicationNode:** An application using the framework and sending work to it. Usually connected to a single master (or multiple masters on a multi-master setup).

NODES CONT'D


BaseNode
Extends
Extends
ServerNode
Node
Extends
Extends
ComputeNode
Extends
Extends
MasterNode
SlaveNode
ApplicationNode
Extends
Implements
Extends
Extends
Implements
Implements
Component
IApp

TASK SCHEDULING

Concurrent comes with two task scheduling policies, one optimized for heterogeneous systems and another for homogeneous systems.

- **Generic:** For heterogeneous systems. Sends work to the best slave for the given work. Comes with slightly more overhead.
- **ZMQ:** ZMQ push/pull scheduling, for homogenous systems. Slave requests task from a global socket. Less overhead but prone to stalls if hardware is not the same on all slaves.

TASK SCHEDULING CONT'D

TASK SCHEDULING CONT'D

From previous slide

Generic scheduling execution flow. A *GenericTaskScheduler* uses a *GenericTaskScheduleStrategy* to send work to a *GenericTaskManager* of the target slave.

TASK SCHEDULING CONT'D

TASK SCHEDULING CONT'D

From previous slide

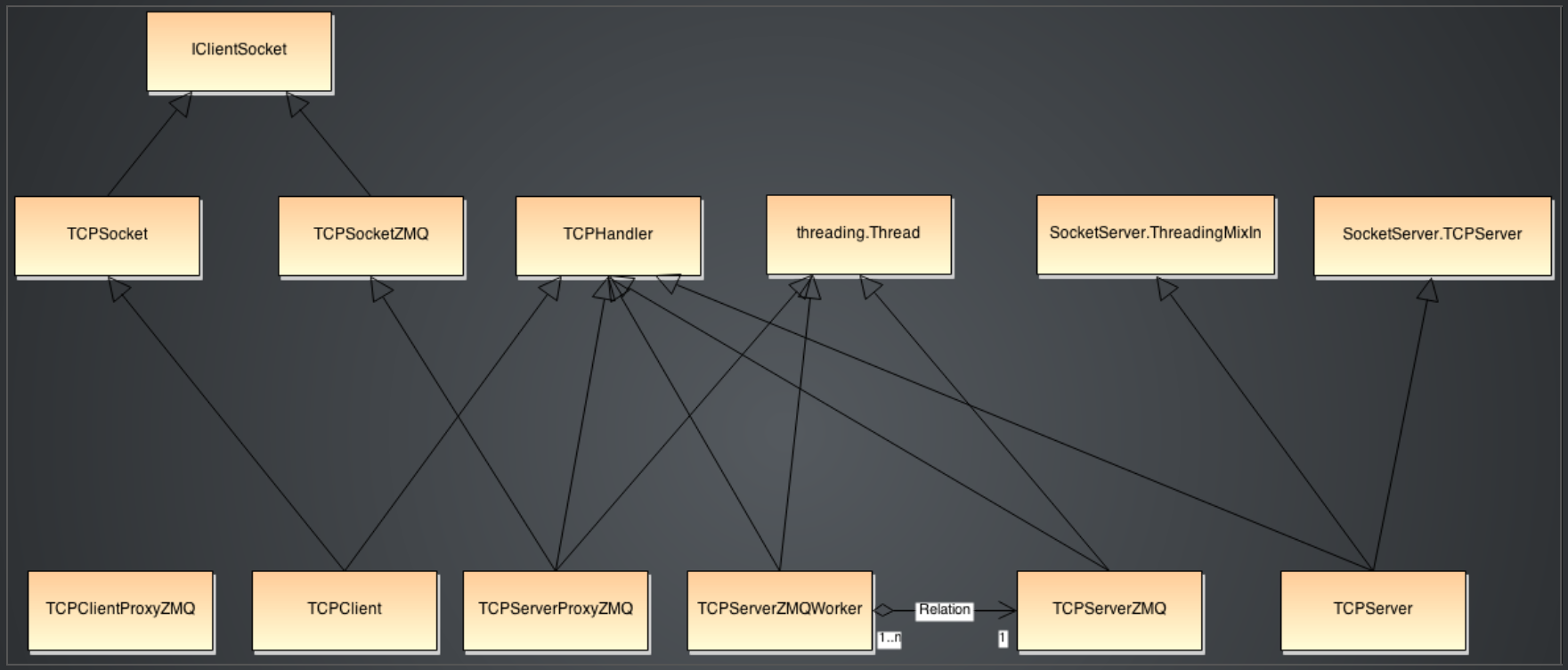
ZMQ push/pull scheduling execution flow. A *ZMQTaskScheduler* pushes work onto a global work socket. The *ZMQTaskManagers* of the slaves pull that work from it and perform the processing. The result is then pushed back to the master node.

TRANSPORT

Concurrent comes with a complex transport module that features TCP and ZMQ sockets clients, servers and proxies.

- **TCPServer:** multithreaded TCP socket server.
- **TCPServerZMQ:** multithreaded ZMQ server using a limited number of pooled workers.
- **TCPClient:** TCP client used to establish connection to a *TCPSocket*.
- **Proxies:** proxies are used to implement RPC like calling mechanisms between servers and clients.
- **TCPHandler:** container handling registration of RPC methods.

TRANSPORT CONT'D



TRANSPORT CONT'D

Registering RPC methods is straightforward in concurrent. Just register a method with the given server or client instance.

```
@jsonremote(self.api_service_v1)
def register_slave(request, node_id, port, data):
    self.stats.add_avg('register_slave')
    return self.register_node(node_id, web.ctx['ip'], port, data, NodeType.sl

@tcpremote(self.zmq_server, name='register_slave')
def register_slave_tcp(handler, request, node_id):
    self.stats.add_avg('register_slave_tcp')
    return self.register_node_tcp(handler, request, node_id, NodeType.slave)
```

MAIN FEATURES

- **No-GIL:** no GIL on our tasks.
- **Balancing:** tasks are balanced using load balancing.
- **Nice to TCP:** internal buffering to avoid TCP congestion.
- **Deployment:** easy to deploy application onto concurrent.
- **Fast development:** easy application framework to build concurrent applications that work on a high number of machines in minutes.
- **Batching:** task batching to simple task schemes.
- **ITaskSystem approach:** autonomous systems control tasks. Easy to implement concurrency in an organize fashion.
- **Plug-able:** all components are plug-able, flexible in development and favor adding new features.
- **API:** RESTful JSON API and TCP/ZMQ API in the same fashion. From the programmer calling a high-performance TCP method is the same as calling a web-service

FUTURE OF CONCURRENT

- **GPU Processing:** enable GPC task processing.
- **Optimize network congestion:** enable data syncronization and optimize locality of required data.
- **Sandboxing:** include a sandboxing feature so that tasks from different applications do not collide.
- **Security:** add cerificates and encryption layers on the ZMQ compute channel.
- **Statistics and monitoring:** include statistics and real-time task monitoring into the web interfaces of each node.
- **Asyn I/O:** optimize servers to use async I/O for optimal task distribution.
- **Multi-master:** implement a multi-master environment using a DHT and a NCS (Network Coordinate System).

SAMPLES

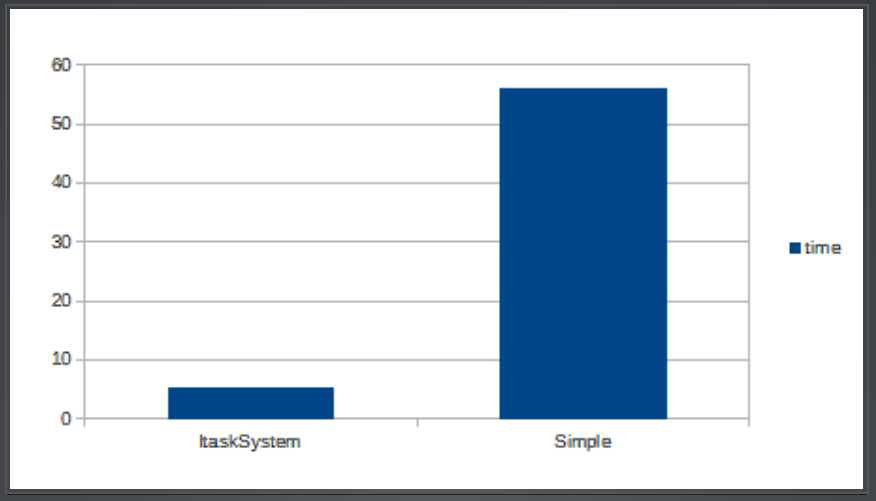
Concurrent comes with a set of samples that should outline the power of the framework. They also guide how an application for concurrent should be created and what has to be avoided.

- Mandelbrot
- Benchmark
- MD5 hash reverse
- DNA curve analysis

MANDELBROT SAMPLE

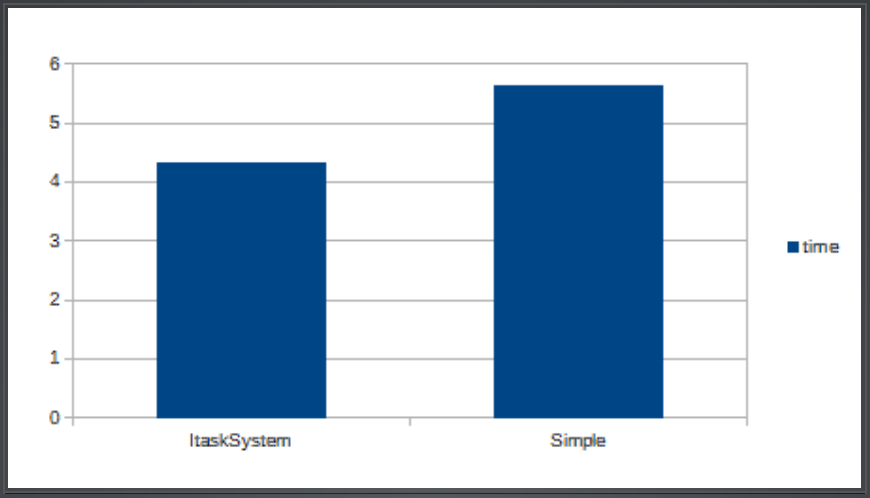
Sample implanted using plain tasks and an *ITaskSystem*. Comes in two flavors of tasks, an optimized and a non-optimized tasks on the data usage side.

MANDELBROT SAMPLE CONT'D



Execution time between an *ITaskSystem* and a plain task implementation using the non-optimized tasks.

MANDELBROT SAMPLE CONT'D

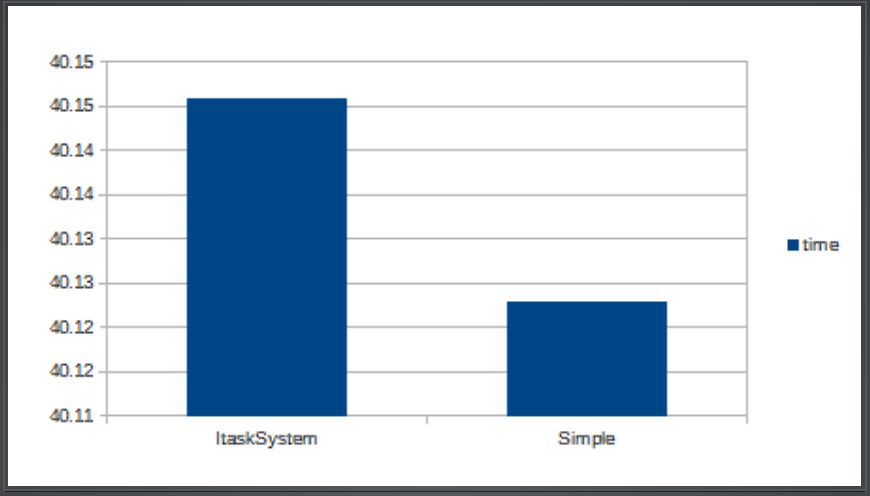


Execution time between an *ITaskSystem* and a plain task implementation using optimized tasks. Both ways gain but the plain tasks experience the main boost.

BENCHMARK SAMPLE

Sample using plain tasks and an *ITaskSystem*. A simple active wait task used to compare the framework and its setup to Amdahl's Law.

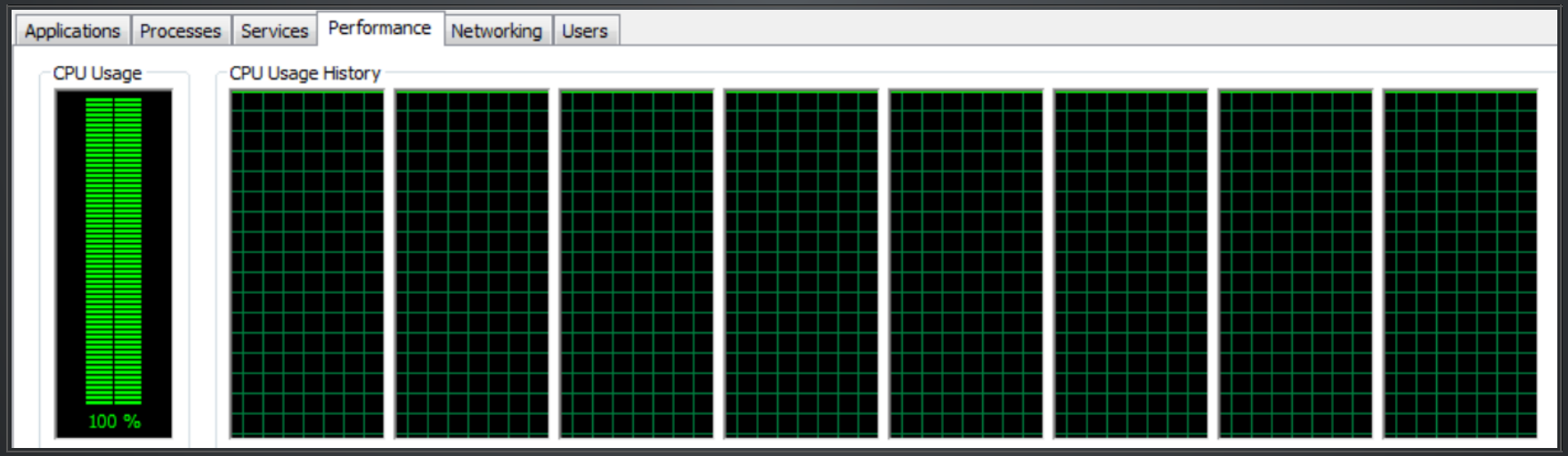
BENCHMARK SAMPLE CONT'D



Execution time between an *ITaskSystem* and a plain task implementation. Both are nearly identical.

BENCHMARK SAMPLE CONT'D

age Name	User Name	CPU	Memory (...
thon.exe	Moss	13	21,620 K
thon.exe	Moss	13	21,628 K
thon.exe	Moss	13	21,652 K
thon.exe	Moss	11	21,652 K
thon.exe	Moss	11	21,656 K
thon.exe	Moss	10	21,624 K
thon.exe	Moss	08	50,080 K
thon.exe	Moss	07	48,424 K
thon.exe	Moss	07	21,652 K
thon.exe	Moss	04	21,632 K
thon.exe	Moss	04	46,696 K



CPU usage of the sample indicating that we are using all available CPU power for the benchmark.

MD5 HASH REVERSE SAMPLE

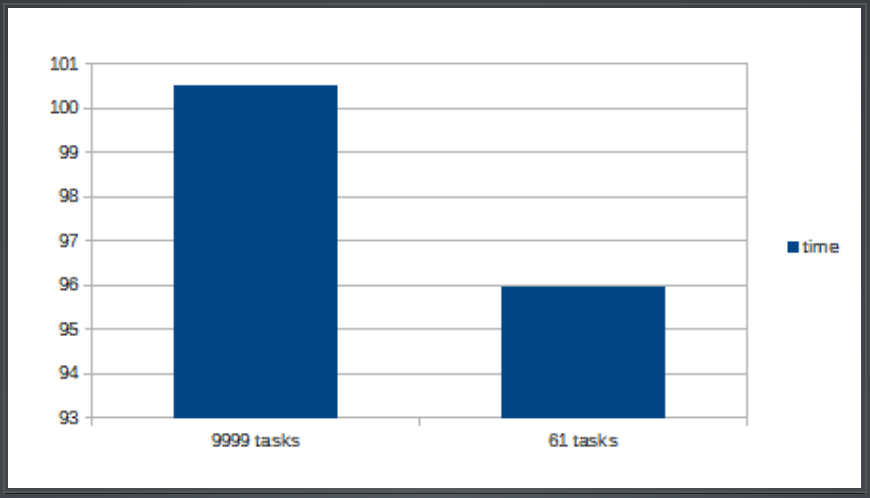
Very simple sample with low data transmission. Uses bruteforce to reverse a numerical MD5 hash knowing the range of numbers. The concurrent implementation is about 4 times faster then the sequential version.

DNA CURVE ANALYSIS SAMPLE

The DNA sample is just a simple way to try to overload the system with over 10 thousand separate tasks. Each task requires a considerable amount of data and so sending it all at once has its drawbacks.

The tests and benchmarks has been performed on a single machine and so we setup the worst possible distributed network resulting in network overloads. This the sample is not intended for real use, it outlines the stability of the framework.

DNA CURVE ANALYSIS SAMPLE CONT'D



As for the other samples the amount of data send through the network is considerable, the sample itself reaches a high memory load up to 3 GB on the master.

Sending huge amounts of data is the real bottleneck of any distributed task framework. We spend more time sending data then performing the actual computation. While in some cases sending less tasks with more data for each one is better then sending thousands of small tasks.

CONCLUSION

- **Complexity:** building a distributed task system is an extremely complex endeavor.
- **Python:** Python has proven to be a perfect choice for its flexibility, ease of development and speed using native modules.
- **ITaskSystem vs plain tasks:** depending on the problem each way fits in its own while in most cases we experienced better results using the *ITaskSystem* approach.
- **Fair network usage:** fair network usage avoiding congestion is vital to maintain a reliable system, sending too much data over the same socket will result in package loss and so in turn in resending of the data.
- **Data locality:** sending data, specially over the MTU (maximum transmission unit) size, has a great impact in the overall performance. Sending less data or performing local reads boosts the processing considerably.

CONCLUSION CONT'D

- **GIL:** threading in Python is a no-go, the GIL impacts heavy on the overall performance. Our master server implementations (TCPServer and TCPServerZMQ) are currently the bottleneck, while this can be addressed using async I/O and processes instead of threads.
- **Sequential vs Parallel:** we achieved a very good parallelization proportion for our samples, the benchmark sample achieved a speedup of about 740% and the Mandelbrot by 175% and 240% depending on the implementation. See next slide for details.

CONCLUSION CONT'D

A main law creating any concurrent computing system is *Amdahl's Law*. Analyzing the performance from our samples, specially the Mandelbrot and the benchmark sample gives is fairly good speed up compared to the maximum speedup that could be achieved applying the law.

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Image courtesy of Wikipedia

- **Mandelbrot:**
  2.4 = 1/((1-P)+(P/N)) where P = 0.67 and N = 8
- **Benchmark:**
  7.02 = 1/((1-P)+(P/N)) where P = 0.98 and N = 8
- **Max Speedup:**
  8 = 1/((1-P)+(P/N)) where P = 1 and N = 8

STELLAR LINKS

- Source code on GitHub
- Project page
- API Documentation
- Video presentation

THE END

BY MORITZ WUNDKE / MORITZWUNDKE.COM

Build with reveal.js