USE OF TORNADO IN KAT-7 AND MeeRKAT FRAMEWORK

C. De Villiers#, B. Xaia‡, SKA South Africa, National Research Foundation of South Africa, South Africa

Abstract

The KAT-7 and MeerKAT radio telescope control systems (www.ska.ac.za) are built on a rich Python architecture. At its core, we use KATCP (Karoo Array Telescope Communications Protocol), a text-based protocol that has served the projects very well. KATCP is supported by every device and connected software component in the system. However, its original implementation relied on threads to support asynchronous operations, and this has sometimes complicated the evolution of the software. Since MeerKAT (with 64 dishes) will be much larger and more complex than KAT-7, the Control and Monitoring (CAM) team investigated some alternatives to classical threading. We have adopted Tornado (www.tornadoweb.org) as the asynchronous engine for KATCP. Tornado, popular for Web applications, is built on a robust and very efficient coroutine paradigm that in turn is based on Python's generators. Co-routines avoid the complexity of thread re-entrancy and lifetime management, resulting in cleaner and more maintainable user code.

This poster will describe our migration to a Tornado co-routine architecture, highlighting the benefits and some of the pitfalls and implementation challenges we have met.

KATCP IN THE KAT-7 AND MEERKAT SYSTEMS

KATCP [1,2] is a simple ASCII communication protocol layered on top of TCP/IP.

It has been developed as a part of the Karoo Array Telescope (KAT) and MeerKAT projects and used at SKA South Africa for the monitoring and control of hardware devices. In this role it has been very successful and the specification is currently at Revision 5.

The original KATCP implementation provided a blocking client and a non-blocking CallbackClient.

Base message types are Request, Reply and Inform - the latter are sent asynchronously by a server to provide out-of-band data or (in some cases) to provide a way of segmenting the results of a previous request.

KATCP additionally defines a software Sensor type. Sensors are created with names and data types such as float, string etc. Dynamically a sensor may acquire a value and a status which it communicates to its registered listeners via callbacks. A listening client may set a strategy on the sensor which causes it to push its value and status to the listener periodically or on certain events, such as a value change. An ad-hoc query mechanism is also available.

KATCP messages, sensors, servers and clients are the building-blocks of the KAT-7 and MeerKAT Control and Monitoring systems. These robust abstractions support the next layer of the architecture, which comprises software proxies to abstract access to real hardware, and components that partition the work of system startup and shutdown, scheduling, observation control and monitoring.

LIMITATIONS OF THREADS

In earlier implementations of KATCP, the inherent concurrency of real-time processes was modelled using software thread. Threads provide the illusion of parallel execution within a single processor core by performing pre-emptive task switches at the system level. This has benefits for processing efficiency since the system need never be idle - while some thread of control is awaiting an event, another thread can be executing. This can be especially useful in networked systems where many cycles would otherwise be wasted waiting for I/O events.

However threads also have some well-known drawbacks. Each thread has its own execution context and this means that threading is resource-intensive, so that the number of active threads must be limited. Perhaps even more important is the complexity they introduce into software design. Since any thread of control may be interrupted or resumed at essentially arbitrary moments, software becomes ‘non-linear’ and the designer must carefully guard against inadvertent corruption of shared resources. Numerous best-practices and software constructs exist to alleviate these problems, but all contribute to the complexity and cost of software development and maintenance.

Finally the Python language, which has proved immensely valuable in the development of our systems, implements a Global Interpreter Lock (GIL) which essentially disables threading for compute-bound tasks on a single processor.

MIGRATION TO TORNADO

Like many teams facing these challenges, we have been interested in the developments in coroutine-based concurrency frameworks. Coroutines are a generalisation of subroutines based on co-operative multitasking, in contrast to the pre-emptive model used by threads Coroutines differ from subroutines in allowing multiple entry-points (and multiple entries per entry-point) within the body of a routine, with the preservation of the full execution context at that point. Because task-switching is

#charles@ska.ac.za
‡ bxaia@ska.ac.za

ISBN 978-3-95450-148-9

cooperative, the developer can tell by inspecting the code where a context switch may occur. This makes it much easier to ensure and verify the determinism of the code. Task switches are 'lightweight' because only the normal stack mechanism is required to save and restore context. All coroutines in a process typically run within a single execution thread.

Coroutines are an important tool to support lightweight concurrency, but for a large system one also needs a scheduling mechanism so that independent subtasks can execute without mutual awareness. This is achieved through a coroutine framework. After some research we chose the open-source Tornado framework.

Tornado[3] is a Python web framework and asynchronous networking library, and offers non-blocking I/O and concurrency support via coroutines. It is capable of scaling to tens of thousands of open connections and concurrent handlers. In addition it provides a scalable, non-blocking Web server and application framework. Although Web development is not our primary focus, the MeerKAT GUI is web-based, and HTTP servers are also proving useful in other areas. Tornado allows for multiple long-lived client connections with minimal overhead.

The MeerKAT GUI displays have been completely re-engineered using Tornado and other modern technologies such as AngularJS. The MeerKAT GUI displays real-time data from the back-end components. Using the Tornado web server and websockets, tests have shown that it can comfortably handle multiple, concurrent, long-lived connections from components as well as human users. Additional libraries and adapters have eased integration, such as toredis (a Redis client on top of Tornado), sockjs_tornado (WebSocket emulation), etc.

KATCP Implementation using Tornado

An objective of our adoption of Tornado was to replace the use of threading throughout the codebase. Because the original KATCP client and server base classes were thread-based, this was the natural starting-point for the implementation.

The Tornado scheduler is called the 'ioloop'. Every component and activity requiring scheduling must have a reference to the ioloop, This reference may be obtained from the global execution context, or passed in as a parameter. The latter method allows for a 'local' ioloop to be used in specific cases.

A particular challenge of the CAM implementation was our large legacy codebase, which has many instances of operations expecting synchronous (blocking) responses. The Tornado ioloop mechanism, however, is non-blocking and returns Futures - placeholders for the result of possibly incomplete operations. Obtaining the result of the operation that returns a Future requires the use of the yield keyword within a coroutine. On encountering this construct, the ioloop engine suspends the current operation and continues with the execution of other coroutines until the result of the Future is available. Then the original stack context is restored and the function continues with the result it has obtained.

KATCP proxies and other top-level components typically run within their own threads or processes to minimize I/O contention. To facilitate the transition from thread-based to coroutine-based concurrency, a compatibility layer was added to KATCP. Some of the classes in this compatibility layer are briefly described below.

The IOLoopManager helper class provides a facade for an ioloop instance that may be shared across components, or running in a private thread. This class exists to abstract this difference and to guard against inadvertent thread contention.

Even when a section of the codebase has been converted to coroutines, we often need a way to return a synchronous result for clients that expect this, and that may be running within their own arbitrary threads. This has been achieved by implementing Python code decorators within our custom compatibility layer: the *_make_threadsafe()* wrapper ensures that arbitrary code is executed within the ioloop's own thread, while the *_make_threadsafe_blocking()* decorator additionally guarantees that the Future's result will be resolved before it is returned to the blocking caller. A DeviceClient instance may call *enable_thread_safety()* before it is started, in order to apply the relevant decorators to all its methods. The whole instance thus becomes implicitly threadsafe and/or blocking, and hence suitable for integration with legacy code.

A feature of KATCP from its origins was introspection; a client can query any device on the network and obtain a specification of its interface (requests and sensors). On connection. the client builds a local representation of the server's interface, and proxies those capabilities using Python's dynamic binding. This feature is implemented in the InspectingClient class. The InspectingClient monitors its own connection and synchronisation state and attempts to re-initialise itself if the connection is lost or the server indicates an interface change. The Inspecting Client in turn may be included in a higher-level container, where it can be commanded and interrogated by an interactive user or a script.

Figure 1 shows a conceptual view of the CAM software layers involved in the Tornado integration.

ISBN 978-3-95450-148-9

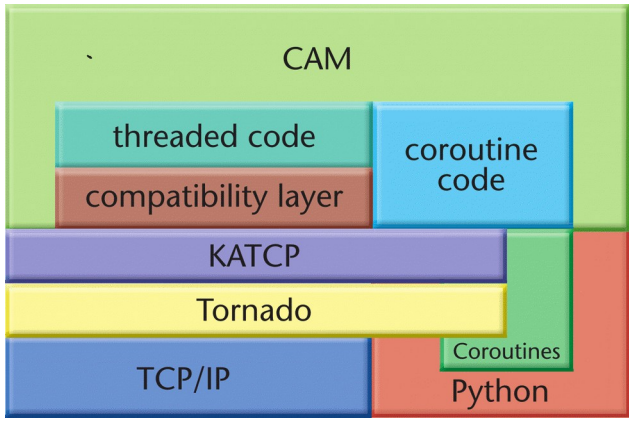

Figure 1: CAM Software Layers.

CONCLUSION

Our adoption of Tornado has brought a number of benefits as well as some challenges.

Benefits

Tornado in CAM has begun to deliver on its promise of efficient multitasking. This is especially evident in areas of our codebase that have been fully converted to the Tornado idiom, such as the sensor-archiving component KatStore and the web-based GUI. Tornado has its own testing framework (built on the standard Python unittest) and this is helping us to eliminate the thread-management challenges that used to hamper our testing. Code at the application level is greatly simplified by the elimination of the complex locking and concurrency control that were necessary for safe threading. This in turn can only benefit reliability and maintainability in the long run.

Challenges

Coroutines and the event loop were an unfamiliar paradigm to most of the team. It takes time to fully understand coroutines and futures, and to recognise the ways in which existing code must change to accommodate them. Despite the compatibility tools added to KATCP, a significant effort has been required to integrate the changes with the rest of the system, and many unit tests initially failed because of explicit or implicit threading dependencies. Some components and some tests have still to be converted.

Debugging can be difficult because of the many layers of 'scaffolding' code introduced by the framework; of course any concurrent model has similar problems. Better tools may help here.

ACKNOWLEDGEMENTS

KATCP was developed as part of the Karoo Array Telescope (KAT) project.

Tornado was developed at FriendFeed.

The Tornado implementation of KATCP was mainly implemented by Neilen Marais as part of the MeerKAT project.

REFERENCES

[1] KATCP documentation,
website: https://pythonhosted.org/katcp/
[2] KATCP GitHub Repository,
website: https://github.com/ska-sa/katcp-python
[3] Tornado documentation,
website: https://tornado.readthedocs.org/en/stable/

ISBN 978-3-95450-148-9