

ETS Tuple space storage patterns

Intent

You need access to data in parallel with fast lookup and need to write data in a serializable way. You also want to avoid blocking on garbage collection.

Motivation

When designing Erlang systems, one of the common operations is to share data between processes. Commonly, there is some data which you need to access from multiple processes simultaneously. One way of handling this is to make a process which governs the data resource. You implement a protocol for reading and writing data to that process. But this solution has poor scalability. The governing process can only satisfy a single reader or a single writer at a time — and this means our solution is not parallel.

What we would like is to avoid having processes blocking on the governing process. To achieve this, we need parallel access to the data structure. However, we still may want to run writes in serial to make sure that we don't accidentally mess up consistency rules of our table.

Another problem is that the governing process might end up having a large heap on which it runs garbage collection. When the process is garbage collecting, it can not satisfy queries. This incurs latency on the application because it makes responses slow.

Context

Erlang has a system called ETS, the Erlang Term Storage. ETS provides what is known as a tuple-space for storing and retrieving Erlang tuples. The key properties of ETS are:

- Access to ETS can happen in parallel with several processes accessing at the same time. Current scalability is around 64 cores.
- Access is fast. Usually measured in microseconds or sub-microseconds.
- Data stored in ETS is manually collected and is stored outside the process heaps. Thus, data residing in ETS does not impose garbage collection overhead, nor does it result in large pause times for processes.

An ETS table has an owning process, which by default is the process creating the table. Tables can be handed off to other processes upon termination of the current controlling process. There are three possible access modes: `private` (only the owner can access), `protected` (only the owner can write, but everyone can read) and `public` (reads and writes can be done by any process). The default mapping is 'protected'.

Our solution design is this: A governing process creates an ETS table in `protected` mode. It satisfies writes like before, but reads can go directly to the ETS table. This avoids the blocking and parallelizes reads. Also, since ETS is not garbage collected and is kept off the heap of the process, we do not incur the garbage collection overhead for data in ETS. Hence, the time it takes for the process to garbage collect is not noticeable.

Structure

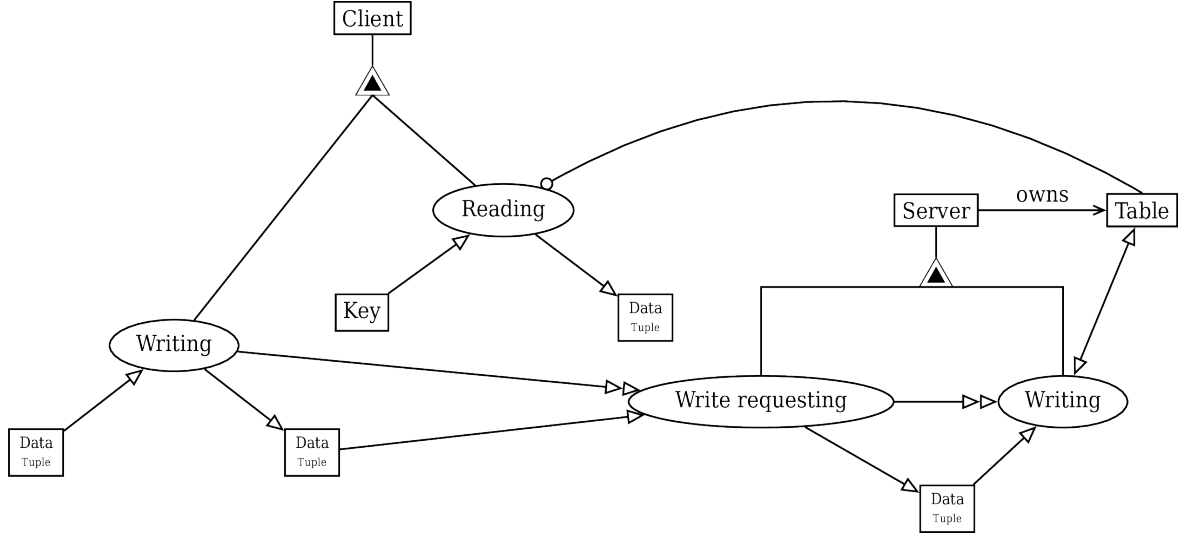


Figure 1: Tuple Space Storage OPM diagram

The Client exhibits Reading and Writing. Reading consumes a Key and uses that to get the corresponding {Key, Value} tuple from Table which is protected by Server. Writing takes a {Key, Value} tuple and invokes Write requesting on the Server, which in turn invokes the internal Writing process that puts the tuple into the table.