Tupelo

Joel VanderWerf
11 Dec 2013

https://github.com/vjoel/tupelo

- Original goal
  - Improve Rinda (ruby stdlib, M. Seki)
- Goal now
  - Maximize trade value of single bottleneck in distributed system

- Three old ideas
  - Tuplespace
  - Transactions
  - Atomic multicast

Tuplespace

- The REST of distributed objects
- Many nouns, few verbs
- Nouns: built from [], {}, string, number, boolean
  - like JSON objects
- Verbs: read(), write(), take()
  - meaning of verb applied to noun depends on noun (as in REST), but constrained
- *Examples*

Tuplespace

- Space abstraction vs channel abstraction
  - Stateful, unlike pubsub
- Decouple endpoints from each other
  - Decouple in space and time
- Decouple application *activities* from OS *processes*

Tuplespace: uses

- Coordination, orchestration
- Concurrent access to shared resources
- Shared config
- Service discovery
- Unique delivery
- Task assignment

Tuplespace: problems

- Locking, bad for concurrency
- Centralization of state and search/wait
- Commercial implementations in legacy mode?

Transactions

- Atomic, isolated change to tuplespace
- Optimistic concurrency, avoid locking
- Avoid lost tuples or need for lease/supervisor
- *Examples*

Transactions in Tupelo

- Mitigate costs:
  - Narrow scope, so execute without 2PC
    - (Limits use of txns across subspaces)
  - Execution in terms of *tuples*, not *templates*
    - (Preparation is in terms of templates)
- Support read/write/take semantics:
  - Deterministic execution order

Atomic multicast

- Bottleneck: message sequence
  - Stamps transactions with increasing “global tick”
- Exactly once, in-order delivery
- Subscriptions to subspaces

Consistency guarantees?

- Linear event history
  - Globally agreement on initial segments
- Operations append to history
  - At message sequencer's global tick (not client's)
- No wall clocks

Consistency limitations

- Guarantee applies to tuple state, not external state (devices, non-tupelo programs)
- Must be connected to message sequencer
  - Bad for availability under network partitions

Latency

- 0 hops to some data
  - subscribed subspace is cached locally
  - read_nowait()
- 2 hops to other data

Replication, durability

- Tupelo separates replication problem into:
- App data replication
  - Use tuplespace ops, subspaces to replicate
  - Use ack tuples for synchronous replication
- Control data replication
  - As hard problem as any replication/failover
  - But state is small
  - Work needed...

Examples

- Map-reduce

```
ruby example/map-reduce/remote-map-reduce.rb
```

- Prime factorization

```
ruby example/map-reduce/prime-factor.rb
```

- Address book with replicated red-black trees

```
ruby example/subspaces/addr-book.rb
ruby example/subspaces/addr-book.rb --show-handlers
```

- Polyglot storage and queries

Summary

- Cost
  - Bottleneck, but probably not SPoF

Summary: benefits

- Distribute computation and data
  - Bring computation to data
- Distribute to any language that talks msgpack
  - Compare riak, rethinkdb
- Pluggable data stores per subspace: key-value or database
- Choice of lock or optimistic concurrency
- Built-in replication of app data
- Built-in load balancing
  - Framework for expressing better or worse algorithms
- Built-in caching / cache invalidation
- Clear model for answering consistency questions

Summary: use cases

- Redis-like uses
- Resque-like uses
- Coordination and control
  - Not for large data volume
  - Not for logs, streams, binary blobs, etc.
- Groups of related, clustered processes
  - Not for dispersed storage/computation
- Sandbox for dist algo design

Summary: plans

- Languages
  - Sooner: C, elixir/erlang
  - Later: python, go, jvm
- Ecosystem of *tuplets*
  - Subspace-mountable units of computation/storage
  - Dynamic allocation of tuplets
- UDP multicast
- SPoF → just a bottleneck (replication is hard!)

Thanks

- Martha Bridegam, for the reindeer accessories
- Raja Sengupta, for pointing me to tuplespaces