MeshUp

And other Riak hacks

Klarna

- Simplify buying online
- Founded in 2005
- 7.5M users in the Nordics, Germany, Austria, the Netherlands
- >2B USD in transactions this year
- 700KLOC Erlang system
- 80ish Erlang developers

The Product

- Hand-crafted vintage XML-RPC API
- Pretty standard payments-as-a-service
- Except that Klarna assumes the risk for both sides without requiring users to have an account
- Users need to be identified and scored

The Workload

- Low volume, high latency
- High complexity
- Must always accept purchases
- Must not lose accepted purchases

Mnesia

- Has worked so far
- But we're really hitting its limits
- Started looking at alternatives in 2011

Riak

- Always writable :-)
- Configurable replication :-)
- Reasonable latency :-)
- Single-object guarantees only :-(
- Eventual consistency :-(

Migration

- Need a replacement for Mnesia transactions
- Somewhat ambitious refactoring needed anyway
- How do we want our application to look?

MeshUp

- Toolkit for writing applications which separate business logic from effects
- Enable functional programming in the presence of a shared database
- Essentially an interpreter for a DSL
- Call it a workflow engine to make people less worried

Basic Idea

- Computation = workflow = series of methods to call
- Data = snapshot of database = dictionary threaded through the calls

Details

- MeshUp provides only policy, no mechanism
- To execute an operation, the engine calls out to implementations of three foundational interfaces
  - meshup_endpoint declares the workflow for a specific operation
  - meshup_service adapts pure Erlang code to the MeshUp calling conventions
  - meshup_store abstracts side effects/database

Endpoints

```
-module(my_endpoint).
-behaviour(meshup_endpoint).
-export([flow/0]).
```

Workflows

Step 1
Step 2
...
Step N

Workflows

[Step1,
Step2,
...
StepN]

Workflows

[{service1, method1} = Step1,
{service2, method2} = Step2,
...
{serviceN, methodN} = StepN]

Workflows

- Arguments implicit

- Unix pipes

Services

```
-module(my_service).
-behaviour(meshup_service).
-export([call/2, describe/2]).
```

Methods

```
call(my_method, Ctx) ->
  my_mod:my_fun(
    meshup_contexts:get(Ctx, ...),
    ...);
...

describe(my_method, input) ->
  ...;
describe(my_method, output) ->
  ...;
...
```

Contracts

Name 1
Name 2
...
Name M

Contracts

[Name1,
Name2,
...
NameM]

Contracts

[[namespace1, bucket1, key1] = Name1,
[namespace2, bucket2, key2] = Name2,
...
[namespaceM, bucketM, keyM] = NameM]

Contracts

[{[namespace1, bucket1, key1], [{store, my_store}]},
  {[namespace2, bucket2, key2], [{store, your_store}]},
  ...
  [namespaceM, bucketM, keyM]]

Contracts

- Actually, arbitrary term structure
- Dynamic contracts via the MeshUp pattern matcher

Stores

- Read, write, delete
- Representation
- Conflict resolution

Stores

```
-module(my_store).
-behaviour(meshup_store).
-export([put/2, get/1, del/1]).
-export([bind/2, return/3]).
-export([merge/3]).
```

Stores

- Meaning of names defined by what stores do with them
- Naming convention ~ query language

API

```
{ok, Ctx} = meshup:start([{endpoint, my_endpoint},
                          {input, Input}])
```

Basic Idea

- Computation = workflow = series of methods to call
- Data = snapshot of database = dictionary threaded through the calls

Execution Model

```
Ctx0 = Input,
Ctx1 = read(Ctx0, InContract1),
Res1= compute(Method1, Ctx1),
Ctx2 = write(Ctx1, Res1, OutContract1),
...
ok     = commit(Ctx),
Ctx.
```

Data Context

- Reads are cached in the context
- Writes are buffered in the context

Read guarantees

- Read-your-writes
- Reads idempotent

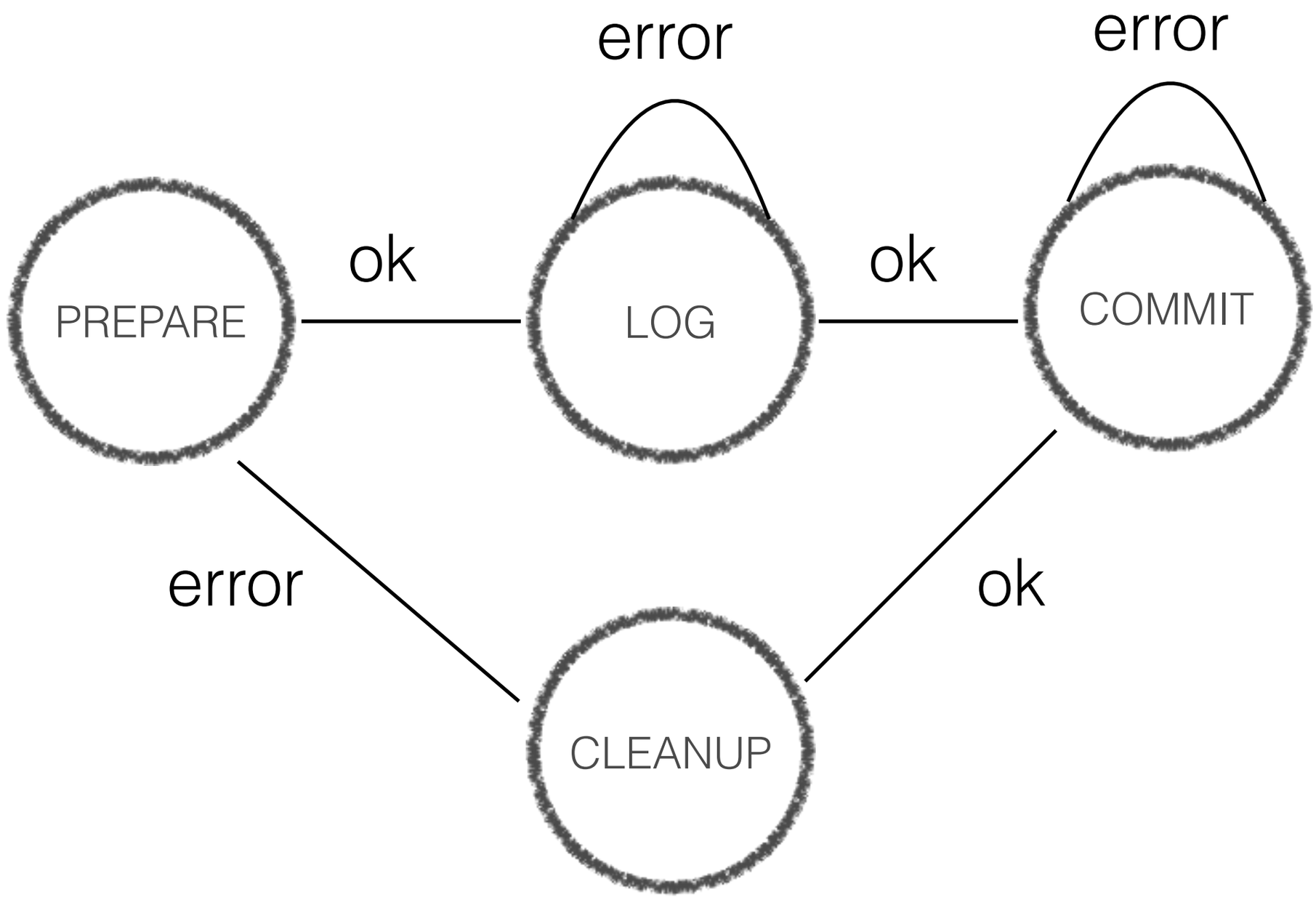
Write guarantees

- Atomic durability for the write-set associated with a workflow
- Failed writes won't show up in the database

Writing to Riak reliably

- Special Riak bucket used as WAL
- Per-node disk_log tracks state of commit

FSM



Custom commit strategies

- Pluggable session store
- Pluggable logger
- Commit mode "write_set"

Data consistency

- Exploit ability to hook into all reads and writes cleanly to make it as easy as possible to write sound eventually consistent programs

Composable Resolvers

- meshup_resolver behaviour
  - from_fun/1, to_fun/1, compose/2
  - Used in meshup_stores' merge/3 callbacks

Preemptive Conflict Resolution

- Whenever a service updates a value which is associated with some store, MeshUp attempts to merge/3 the new and old values

- Can enable/disable on a per-key basis by matching on the first argument of merge/3

Misc Features

- Session handling
- meshup:transaction(Fun, ReadSet, WriteSet)
- Global read/write policies
- Interactive shell
- Linter
- Promises

Future Work

- Read scheduling
- Post-mortem debugger
- Lazy contexts
- Caching annotations

Supporting code

- KRC - simple Riak client
- RiMU - implementation of MeshUp interfaces for Riak

Conclusion

- "All problems in computer science can be solved by another level of indirection." -David Wheeler

- "Every system is either an interpreter or a compiler." -Don Stewart

Q & A