Seed Tutorial

1 Prerequisites

- Go `http://golang.org`
- Gnuplot
- Seed `https://github.com/nathankerr/seed`
- A web browser (tested in Safari, Chrome, and Firefox)
- Some knowledge of command lines, go, javascript, html, and services.

2 Getting Started

Make a new directory to work in:

```
mkdir seed-tutorial; cd seed-tutorial
```

Create a file, kvs.seed, with the following contents (don't include the line numbers).

```
1 table kvstate [key] => [value]
```

This file defines the core of a key-value store service. Line 1 describes a table called kvstate with a single key column called key and a single value column called value.

3 Starting the Service

Start the service by running:

```
seed -sleep 0.5s -monitor :8000 -execute kvs.seed
```

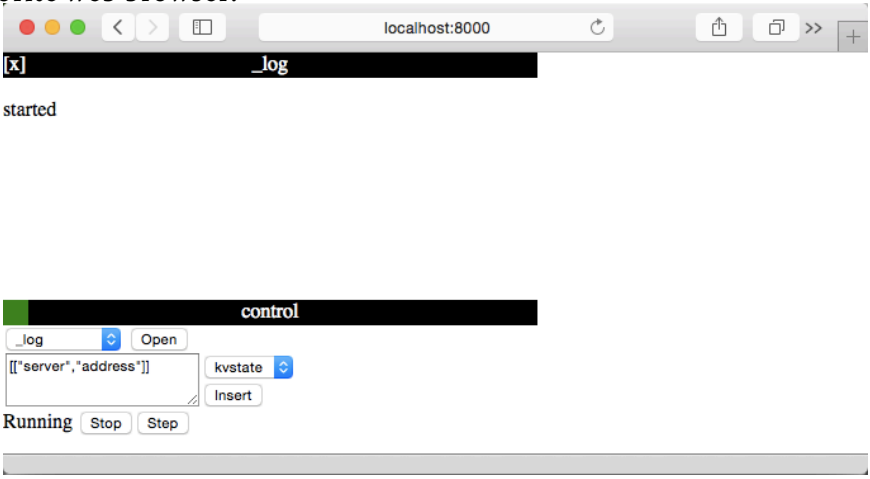
-sleep 0.5s will sleep half a second in between time steps. This slows things down enough so that we can see what is happening in the service instance. A sleep value is needed when using the monitor to limit the data the web browser must process.

–monitor :8000 starts the debugging monitor on port 8000. To restrict access to localhost, use –monitor localhost:8000.

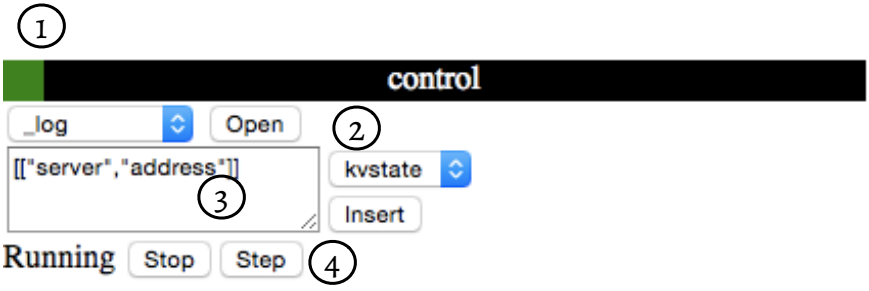
–execute starts the service using the built in interpreter.

4 The Monitor

The monitor can be accessed at `http://localhost:8000` using your favorite web browser:



The debug monitor presents a view of the state of the service instance. The control area has controls for controlling the monitor.



1. Green shows the monitor is connected; red disconnected.
2. Access to available views. Select the desired view and press open.

3. Insert data into a collection. Data is encoded in JSON as an array of arrays. "server" and "address" are special values that are translated into the server and client addresses respectfully.

4. Execution control. Start and stop the service, and step through the parts of the time step.

The rest of the monitor area is for the views. Click on the name of the view to put it in the main area (on the left, where _log is); [x] to close the view. Views beginning with underscores are special.

_log The monitor log

_service The service that is running (expressed in Seed)

_graph The flow graph of the running service. Rule numbers in the graph correspond to those in _service.

_time The time spent on the last timestep

Views not beginning with underscores correspond to collections (e.g., tables, inputs, outputs) in the service.

You can use the monitor to insert data into kvstate. The displayed order of rows for kvstate might fluctuate. This is due to the unordered nature of collections[1].

5 KVS Put Operation

Update kvs.seed to match:

```
1 table kvstate [key] => [value]
2
3 input kvput [key] => [value]
4 kvstate <+- [kvput.key, kvput.value]
```

[1]Collections are implemented using Go's map datatype. When Go maps contain a small number of keys, the order is randomized when accessing all the keys so that developers will not accidentally depend on the implementation's natural ordering of this unordered dataset.

Line 3 defines an input called kvput that contains two columns: key and value. Like all collections, inputs have key and data columns. Input and output collections define the flow of data into and out of a service. These collection types are not network accessible.

Line 4 is a rule which projects the columns in kvput and then upserts (<+−) the resulting set into kvstate.

```
seed -sleep 0.5s -monitor :8000 -execute kvs.seed
```

Insert stuff into kvput and watch how it flows to kvstate. Try using various values to see how things work. Also look at the _service and _graph views to see how they relate. Note that no transformations have been applied (we will transform the service in later steps).

6 Put Client

We will now build a client for the service. When starting the service, add the network interface by using the network transformation:

```
seed -sleep 0.5s -monitor :8000 -execute
   -transformations network kvs.seed
```

Check the service's _service and _graph views to see the results of the transformation.

The client will use a custom json over webservices protocol. This protocol was implemented to allow web browser based clients.

The client is implemented as a static web page (index.html):

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Put Client</title>
5  <meta charset="utf-8" />
6  <script>
7    var websocket, address, connected, server
8    var key, value
9
10   function onClose() {
11     connected.style.backgroundColor = "red"
12   }
13
14   function send() {
```

```
15      try {
16        websocket.send(JSON.stringify({
17          Operation: "<~",
18          Collection: "kvput",
19          Data: [[server, key.value, value.value]]
20        }))
21      } catch (e) {
22        alert(e)
23      }
24    }
25
26    function init() {
27      connected = document.getElementById("connected")
28
29      key = document.getElementById("key")
30      value = document.getElementById("value")
31
32      // connect to the monitor server
33      server = "ws://" + window.location.hostname +
            ":3000/wsjson"
34      websocket = new WebSocket(server);
35      websocket.onclose = onClose;
36
37      // set uniq id (address)
38      websocket.onopen = function() {
39        address = Math.random().toString(36).substr(2)
40        try {
41          websocket.send(JSON.stringify(address))
42        } catch (e) {
43          alert(e)
44        }
45
46        connected.style.backgroundColor = "green"
47      }
48    }
49
50    window.addEventListener("load", init, false)
51  </script>
52  </head>
53  <body>
54  <div id="connected" style="width:20px; height:20px;
        background-color:red"> </div>
55  <div>
```

```
56    <form onsubmit="return false;">
57      <input type="text" id="key" value="Key" />
58      <input type="text" id="value" value="Value" />
59      <input type="submit" value="Put"
            onclick="send()"/>
60    </form>
61  </div>
62  </body>
63  </html>
```

The client is initialized with the init function on Lines 26-48 this is ran on load (Line 50). A websocket connection is setup in Lines 32-35. A pseudorandom address is then created and sent to the server. This address will be used as the client address for Channel based communication and lasts as long as the websocket.

When initialization is completed, the connection indicator (Line 54) is turned green. When the websocket is closed, the connection will turn red (Line 35, Lines 10-12).

When the put button in the form is pressed (or the form is otherwise submitted), the send function in Lines 14-24 is executed. This function simply builds, serializes, and sends the data over the websocket to the server.

Because web browsers don't like allowing websocket access from pages loaded from the local file system, we need a simple web server:

```
1  package main
2
3  import (
4    "fmt"
5    "net/http"
6  )
7
8  const ADDR = ":4000"
9
10 func main() {
11   http.Handle("/", http.FileServer(http.Dir(".")))
12
13   fmt.Println("Starting Put client on", ADDR)
14   err := http.ListenAndServe(ADDR, nil)
15   if err != nil {
16     println(err.Error())
17   }
```

```
18 | }
```

Start the client web server by:

```
go run server.go
```

And then open your browser to `http://localhost:4000`. As usual, the server will bind to all your computer's network interfaces. To restrict it, change ADDR on Line 8 to localhost :4000.

To see a more advanced client that also handles results, look at the time service client's implementation.

7 Replication

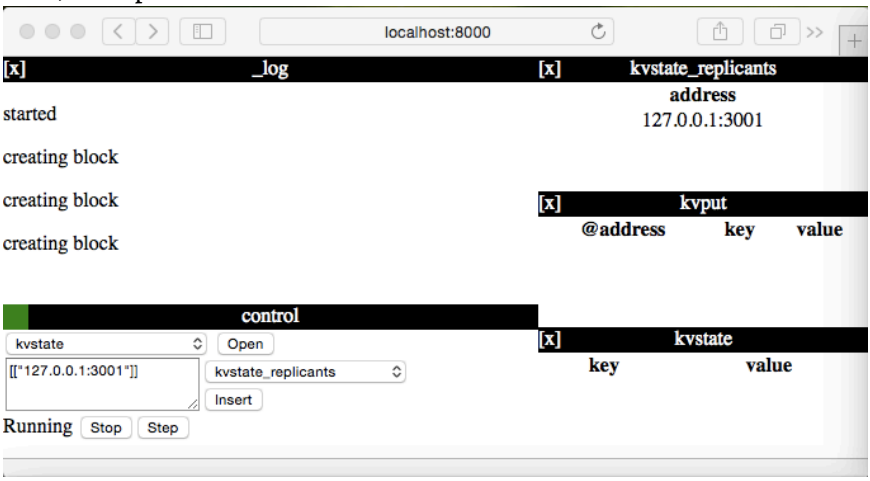
We will now setup and run a replicated kvs. No changes to any code are needed to do this.

Start the first replica:

```
seed -sleep 0.5s -monitor :8000 -execute
    -transformations "network replicate" -address
    :3000 kvs.seed
```

Insert [["127.0.0.1:3001"]] into :8000's kvstate_replicants using the monitor.

Also, set up the views as follows:

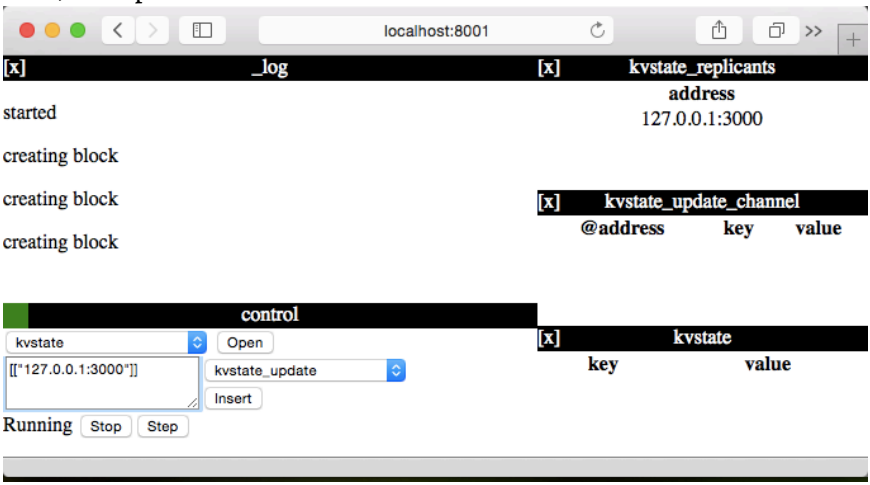


Start the second replica:

```
seed -sleep 0.5s -monitor :8001 -execute
    -transformations "network replicate" -address
    :3001 kvs.seed
```

Insert [["127.0.0.1:3000"]] into :8001's kvstate_replicants using the monitor.

Also, set up the views as follows:



Start a client (will connect to :3000):

```
go run server.go
```

If you have the two replica monitors and the client in different windows, you can watch the entire progression.

Put some value into the client and watch as it first hits the first replia and then the second.

Note that due to the transformations, no changes were needed to kvs.seed or the client to enable replication. However, while the client will only connect to one of the replicas, this could be resolved by using a load balancer.