Work progress

Development of:

- A generic specification template for trace validation (called trace spec.)
- A library (called instrumentation) that enable to log events and changes happening on variables
- A “method” that aims to log implementation properly

3 implementations:

- Two phase protocol (distributed)
- Key value store
- Raft (distributed)

Raft example - spec

Here an example of a base specification (Raft):

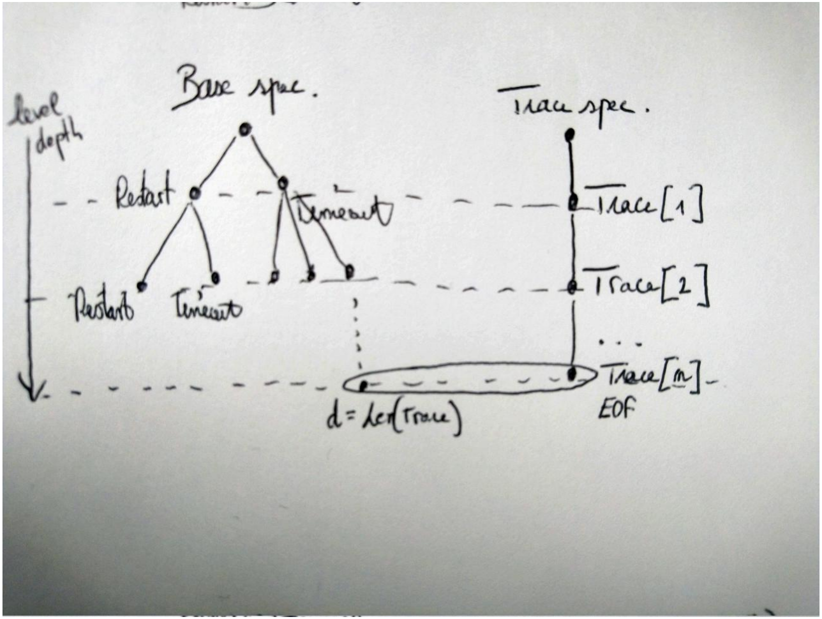
```
\* Defines how the variables may transition.
Next == /\ \/ \E i \in Server : Restart(i)
           \/ \E i \in Server : Timeout(i)
           \/ \E i \in Server : BecomeLeader(i)
...
...

Spec == Init /\ [][Next]_vars
```

Raft example - trace

- A trace can be seen as a behavior of a system
- A trace is a sequence of events (atomic TLA+ action)
- Each event is compound by one or many variable updates
- Below an extract of a trace of raft consensus algorithm

```
{
    "clock": 1,
    "state": [
        {"op": "Replace", "path": ["node2"],
        "args": ["Candidate"]}],
    "desc": "Timeout"
}
...
{
    "clock": 26,
    "state": [{"op": "Replace", "path": ["node1"],
    "args": ["Leader"]}],
    "desc": "BecomeLeader"
}
```

Trace specification - how validate a trace ?



Trace specification - how validate a trace ?

- At least one path of state space graph must lead to the complete reading of trace
- Use a `POSTCONDITION` (hyperproperty)
- Allow TLC to have non-deterministic behavior

```
TraceAccepted ==
    (* Diameter equal to trace length => *)
    (* Trace file has been read completely at least one tin
    LET d == TLCGet("stats").diameter IN
    IF d - 1 = Len(Trace) THEN TRUE
    ELSE Print(<<"Failed matching the trace to (a prefix of
    "TLA+ debugger breakpoint hit count " \

POSTCONDITION
    TraceAccepted
```

Trace specification - how it work ? - spec refinement

- We have to write a trace spec that is a refinement of a base spec (here Raft)

```
(* Temporal formula for trace spec *)
TraceSpec == TraceInit /\ [][TraceNext]_<<l, vars>>

(* Instanciate raft *)
BASE == INSTANCE raft
BaseSpec == BASE!Init /\ [][BASE!Next \/ ComposedNext]_BASE

SPECIFICATION
    TraceSpec
PROPERTIES
    (* Refine raft *)
    BaseSpec
```

Trace specification - read trace events

- Read trace line after line (each line is an event)
- Apply all operations, on all variables found in each events

```
logline == Trace[l]

ReadNext ==
    (* depth: line number *)
    /\ l' = l + 1
    (* Apply all variable updates *)
    /\ MapVariables(logline)
    (* Advance base spec *)
    /\ BaseSpec::Next
```

Trace specification - variables update and mapping

- TLC apply all operations to all variables precised in current event

```
MapVariables(logline) ==
    /\
        IF "state" \in DOMAIN logline
        THEN state' = ExceptAtPaths(state, "state", loglin
        ELSE TRUE
    /\
        IF "currentTerm" ...
```

Note: If a variable changes isn't logged, `TraceSpec` just let TLC search for all possible values of this variable according to base spec (see TRUE).

Trace specification - variables update and mapping

- Variable updates was made by applying 1 or more operators on it
- Operators are generic and defined in trace spec, for example:

```
Replace(cur, val) == val
AddElement(cur, val) == cur \cup {val}
AddElements(cur, vals) == cur \cup ToSet(vals)
RemoveElement(cur, val) == cur \ {val}
Clear(cur, val) == {}
...
```

Trace specification - variables update and mapping

- following event:

```
{
    "clock": 1,
    "state": [
        {"op": "Replace", "path": ["node2"],
        "args": ["Candidate"]}],
    "desc": "Timeout"
}
```

- should map variable `state` as following:

```
state' = [state EXCEPT !["node2"] = "Candidate"]
```

Trace specification - variables update and mapping

- A variable can be updated partially at a given path

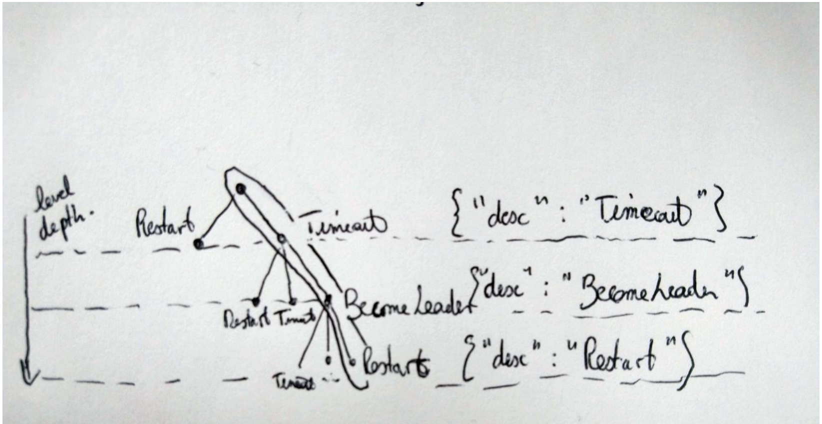
```
{"matchIndex": [{
    "op": "Replace",
    "path": ["node3", "node2"],
    "args": [7]}]}
```

- This update will be automatically translated to:

```
matchIndex' = [matchIndex EXCEPT !["node3"]["node2"] = 7]
```

Trace specification - optimisation

- State space can be largely reduced if we precise the name of the next action expected. Action name lead TLC and select directly the expected action. Therefore reducing non-deterministic behaviors.
- Specify action name when logging is recommended but not mandatory



Trace specification - optimisation

- For each action contained in base spec we write a corresponding predicate
- Predicate enable TLC to select next expected action when `IsEvent` is `TRUE`

```
IsEvent(e) ==
    /\ IF "desc" \in DOMAIN logline
        THEN logline.desc = e ELSE TRUE

IsRestart ==
    /\ IsEvent("Restart")
    /\ \E i \in Server : Restart(i)

IsTimeout ==
    /\ IsEvent("Timeout")
    /\ \E i \in Server : Timeout(i)
...
```

Trace specification - optimisation

- next action of trace spec is just the disjunction of all predicates

```
TraceNext ==
    \/ IsRestart
    \/ IsTimeout
    ...
```

Instrumentation - purpose

- Aims to generate a trace by logging some events
- Aims to log event and variable changes

Trace example:

```
{
    "clock": 1,
    "state": [{"op": "Replace", "path": ["node2"], "args":
    "commitIndex": [{"op": "Replace", "path": ["node2"], "a
    "desc": "Restart"
}
...
```

Instrumentation - How to log

1. We have to log events: log all commits is necessary because TLC cannot fill holes in events
2. We have to log variable changes: log of all variables isn't necessary, but more variables we log, more the statespace reduce, and more we are confident in the implementation

Instrumentation - log event

Example of log “Timeout” event in Raft:

```
public void timeout() {
    assert state == NodeState.Follower;
    ...
    spec.commitChanges("Timeout");
}
```

Instrumentation - log variables

The idea is to log variable updates like you manipulate directly the specification's variables.

Declare spec variable example:

```
this.spec = new TraceInstrumentation(nodeInfo.name() + ".no
// Binding to variable state at path nodeName (state[nodeNa
this.specState = spec.getVariable("state")
                     .getField(nodeInfo.name());
this.specVotesGranted = spec.getVariable("votesGranted")
                     .getField(nodeInfo.name());
```

Instrumentation - log variables

Log variable changes example:

```
private void setState(NodeState state) {
    this.state = state;
    // this.spec.notify(specState, SET, state.toString());
    specState.set(state.toString());
}
...
if (m.isGranted()) {
    // Add node that granted a vote to me
    candidateState.getGranted().add(m.getFrom());
    specVotesGranted.add(m.getFrom());
}
```

Instrumentation - clocks

We can use two way to sync clock between distributed processes:

- Lamport clock, we send clock in the message and we call explicitly sync method on logging framework
- Shared clock, if all the system is executed on the same physical machine, all process can share a clock in a memory mapped file: `SharedClock.get(clockName);`

Execution pipeline

In all our tests we make a script execution pipeline that do the following:

- Execute implementation (which create a trace file by logging events and variable updates)
- Merge trace files that was produced by different processes
- Execute TLC on the trace spec for a given trace file in order to make validation