Testing Asynchronous Behaviour in an Instant Messaging Server

John Hughes

Chalmers University/Quviq AB

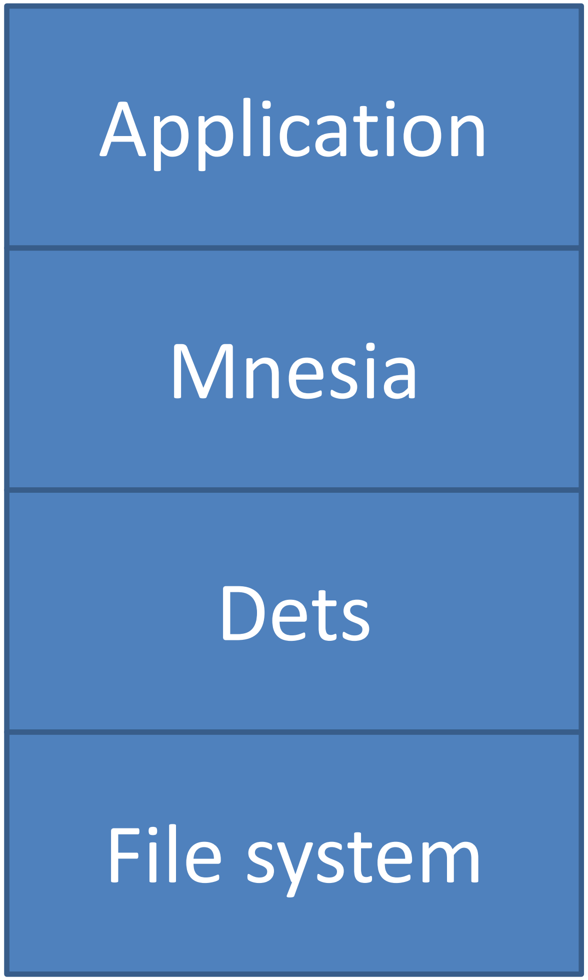
"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year. We have not been able to track the bug down since the dets files is repaired automatically next time it is opened.“

Tobbe Törnqvist, Klarna, 2007

What is it?

300 people in 5 years

klarna



Application

Mnesia

Dets

File system

Invoicing services for web shops

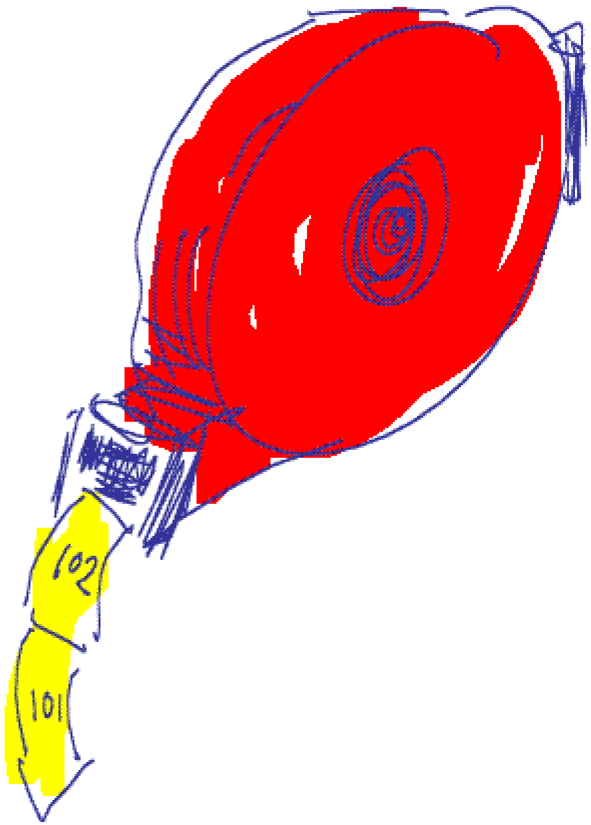
Distributed database: transactions, distribution, replication

Tuple storage

Imagine Testing This…

dispenser:take_ticket()

dispenser:reset()

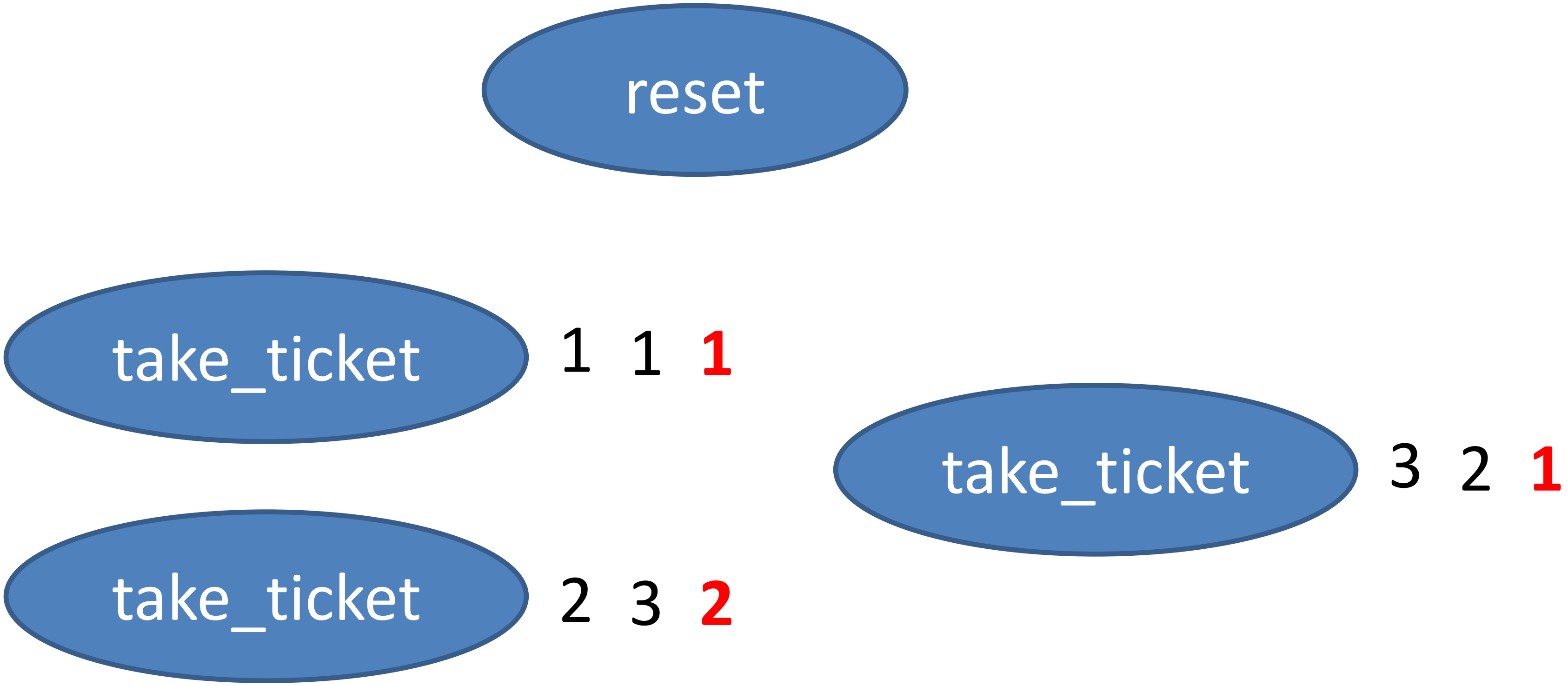


A Unit Test in Erlang

```
test_dispenser() ->
    ok = reset(),
    1  =  take_ticket(),
    2  =  take_ticket(),
    3  =  take_ticket(),
    ok =  reset(),
    1  =  take_ticket().
```

Expected results

A Parallel Unit Test



reset

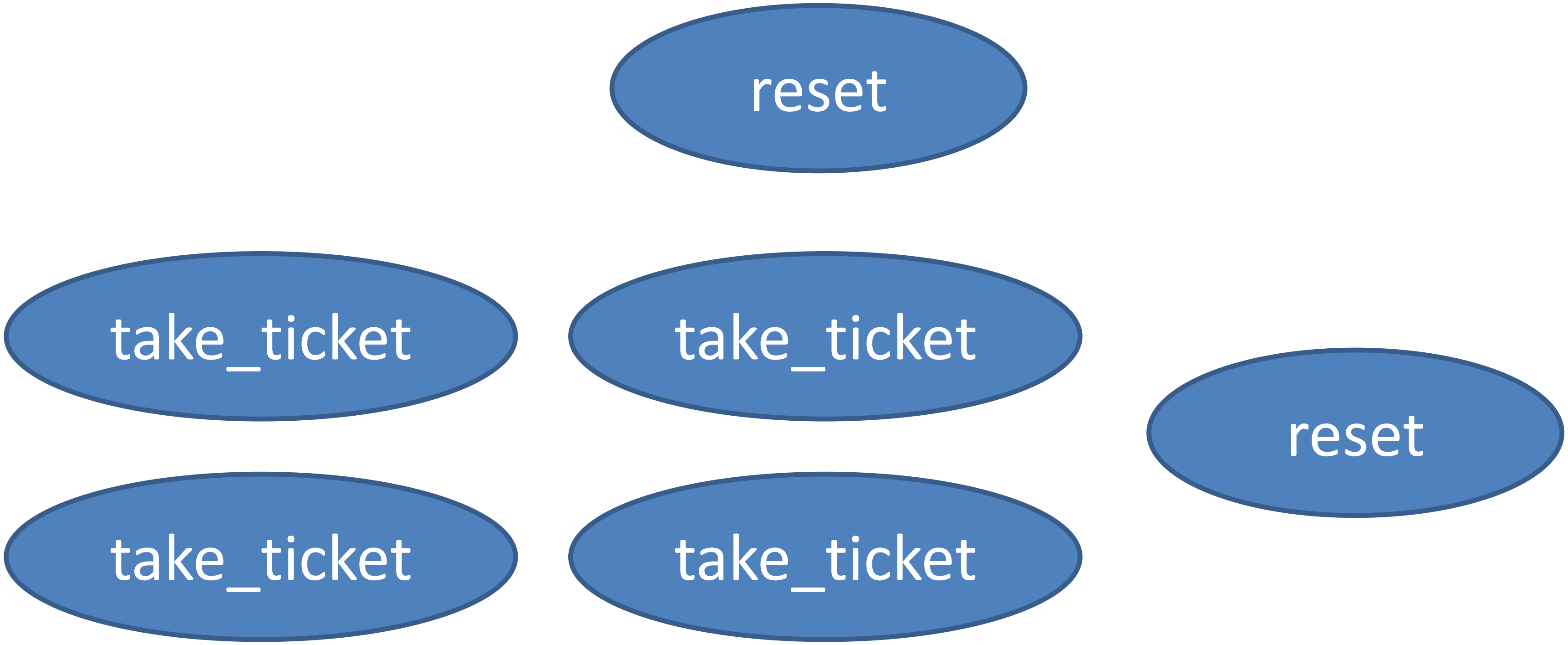
take_ticket 1 1 **1**

take_ticket 3 2 **1**

take_ticket 2 3 **2**

- Three possible correct outcomes!

Another Parallel Test



reset

take_ticket

take_ticket

reset

take_ticket

take_ticket

- 42 possible correct outcomes!

Property-Based Testing

- Write *properties* instead of expected outputs
  - e.g. sort([A,B,C]) == [1,2,3]

- Can handle a *variety* of outputs

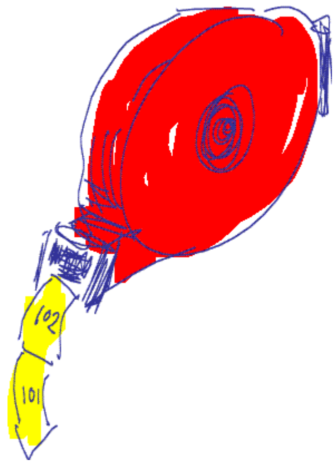
  ➔ can *generate* test cases

QuickCheck Demo

State Machine Models

- Test case is a *list of commands*

  {call,Module,Function,Arguments}



- Model the state abstractly

```
next_state(S,_V,{call,_,reset,_}) ->
    0;
next_state(S,_V,{call,_,take_ticket,_}) ->
    S+1.
```

- Define postconditions

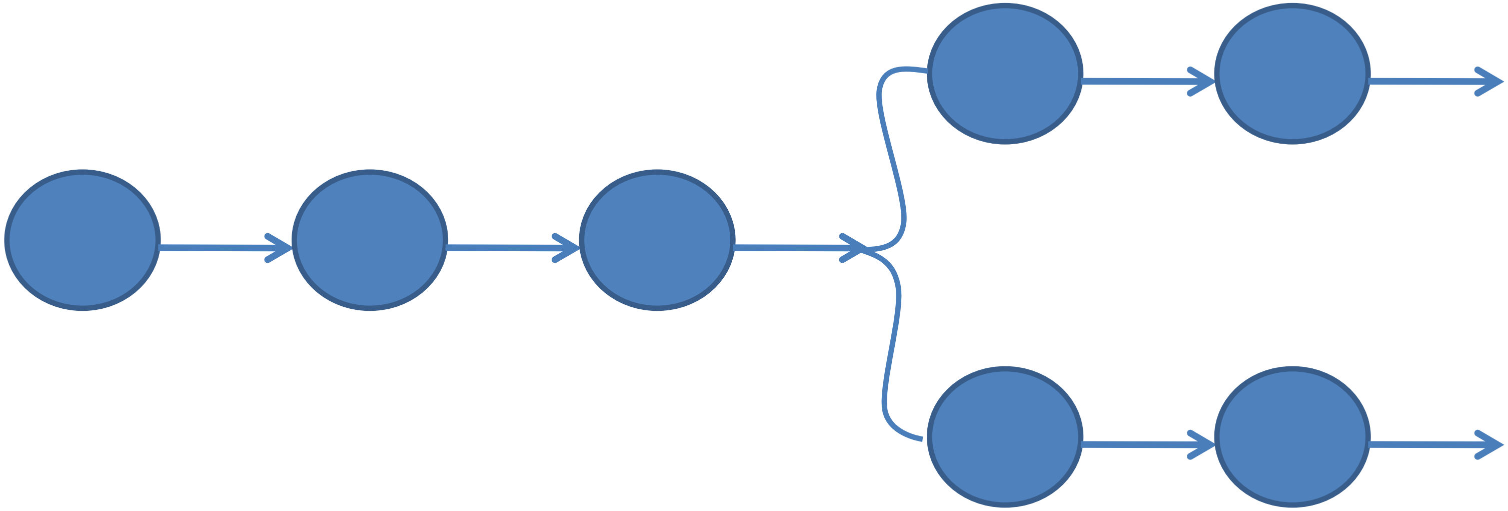
```
postcondition(S,{call,_,take_ticket,_},Res) ->
    Res == S+1;
```

Generate a test case from the callbacks in ?MODULE

```
prop_dispenser() ->
  ?FORALL(Cmds,commands(?MODULE),
    begin
      start(),
      {_H,_S,Res} = run_commands(?MODULE,Cmds),
      Res == ok
    end).
```

Run the list of commands and check postconditions wrt the model state

Parallel Test Cases



- Use the *same* state machine model!

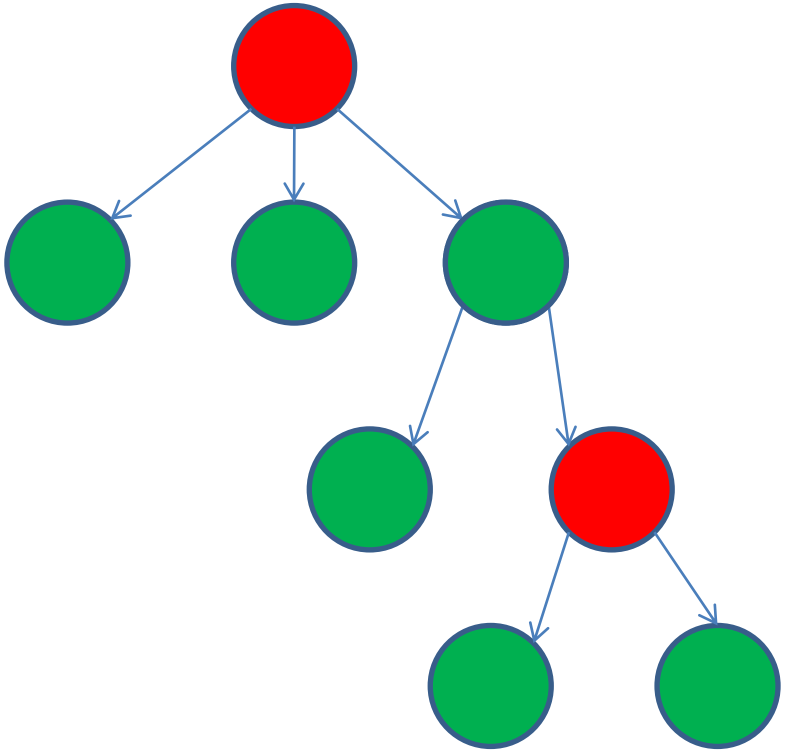
Generate parallel test cases

```
prop_parallel() ->
  ?FORALL(Cmds,parallel_commands(?MODULE),
    begin
      start(),
      {H,Par,Res} =
         run_parallel_commands(?MODULE,Cmds),
      Res == ok)
    end)).
```

Run tests, check for a matching serialization

DEMO

- Sometimes:



```
Prefix:
    take_ticket() --> 1
    reset() --> ok
    reset() --> ok
    reset() --> ok
    take_ticket() --> 1
    take_ticket() --> 2
    reset() --> ok
    take_ticket() --> 1

Parallel:
1.  take_ticket() --> 2
    take_ticket() --> 3

2.  take_ticket() --> 2

Result:
no_possible_interleaving
```

```
take_ticket() ->
    N = read(),
    write(N+1),
    N+1.
```

Prefix:

Parallel:

1. take_ticket() --> 1

2. take_ticket() --> 1

Result: no_possible_interleaving

dets

- Tuple store:

  {Key, Value1, Value2…}

- Operations:
  - insert(Table,ListOfTuples)
  - delete(Table,Key)
  - insert_new(Table,ListOfTuples)
  - …
- Model:
  - List of tuples

200 LOC
vs.
6.3 KLOC

Bug #1

insert_new(Name, Objects) -> Bool

Types:

Name = name()

Objects = object() | [object()]

Bool = bool()

```
Prefix:
   open_file(dets

Parallel:
1. insert(dets_ta

2. insert_new(dets_table,[]) --> ok

Result: no_possible_interleaving
```

Bug #2

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table

Parallel:
1. insert(dets_table,{0,0}) --> ok

2. insert_new(dets_table,{0,0}) --> …time out…
```

```
=ERROR REPORT==== 4-Oct-2010::17:08:21 ===
** dets: Bug was found when accessing table dets_table
```

Bug #3

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table

Parallel:
1. open_file(dets_table,[{type,set}]) --> dets_table

2. insert(dets_table,{0,0}) --> ok
   get_contents(dets_table) --> []

Result: no_possible_interleaving
```

Bug #4

```
Prefix:
   open_file(dets_table,[{type,bag}]) --> dets_table
   close(dets_table) --> ok
   open_file(dets_table,[{type,bag}]) --> dets_table

Parallel:
1. lookup(dets_table,0) --> []

2. insert(dets_table,{0,0}) --> ok

3. insert(dets_table,{0,0}) --> ok

Result: ok
```

premature eof

Bug #5

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table
   insert(dets_table,[{1,0}]) --> ok

Parallel:
1. lookup(dets_table,0) --> []
   delete(dets_table,1) --> ok

2. open_file(dets_table,[{type,set}]) --> dets_table

Result: ok
false
```

bad object

"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year.”

Tobbe Törnqvist, Klarna, 2007

Each bug fixed the day after reporting the failing case

How come?

- Race conditions are *hard* to unit test
- Testing with properties is powerful!
  - Finds cases noone thinks to test

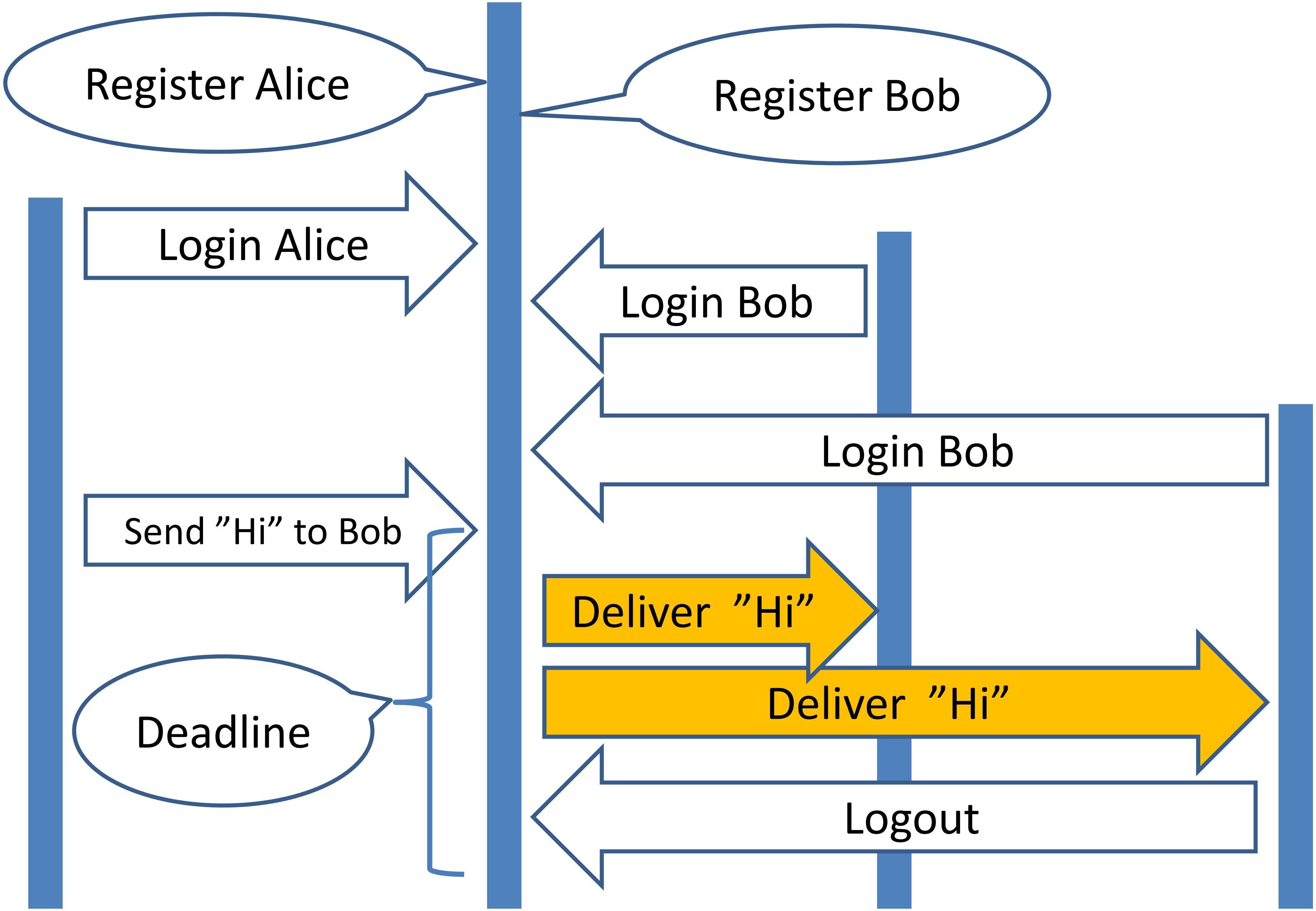
ejabberd

- An instant messaging server

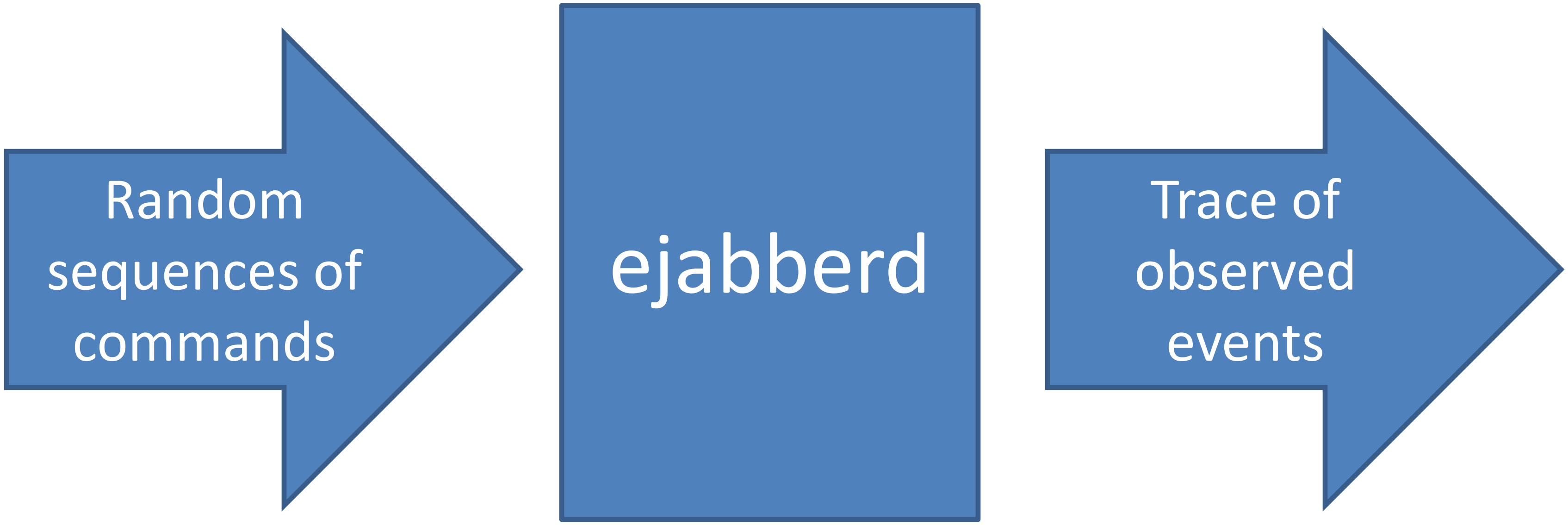
- Market leader in XMPP messaging
  - 38% of XMPP servers run ejabberd

- Improve testing to prepare for a major refactoring
  - In particular, test message delivery

ejabberd


Register Alice
Register Bob
Login Alice
Login Bob
Login Bob
Send ”Hi” to Bob
Deliver ”Hi”
Deliver ”Hi”
Deadline
Logout

Approach



Random sequences of commands

ejabberd

Trace of observed events

Problems, problems

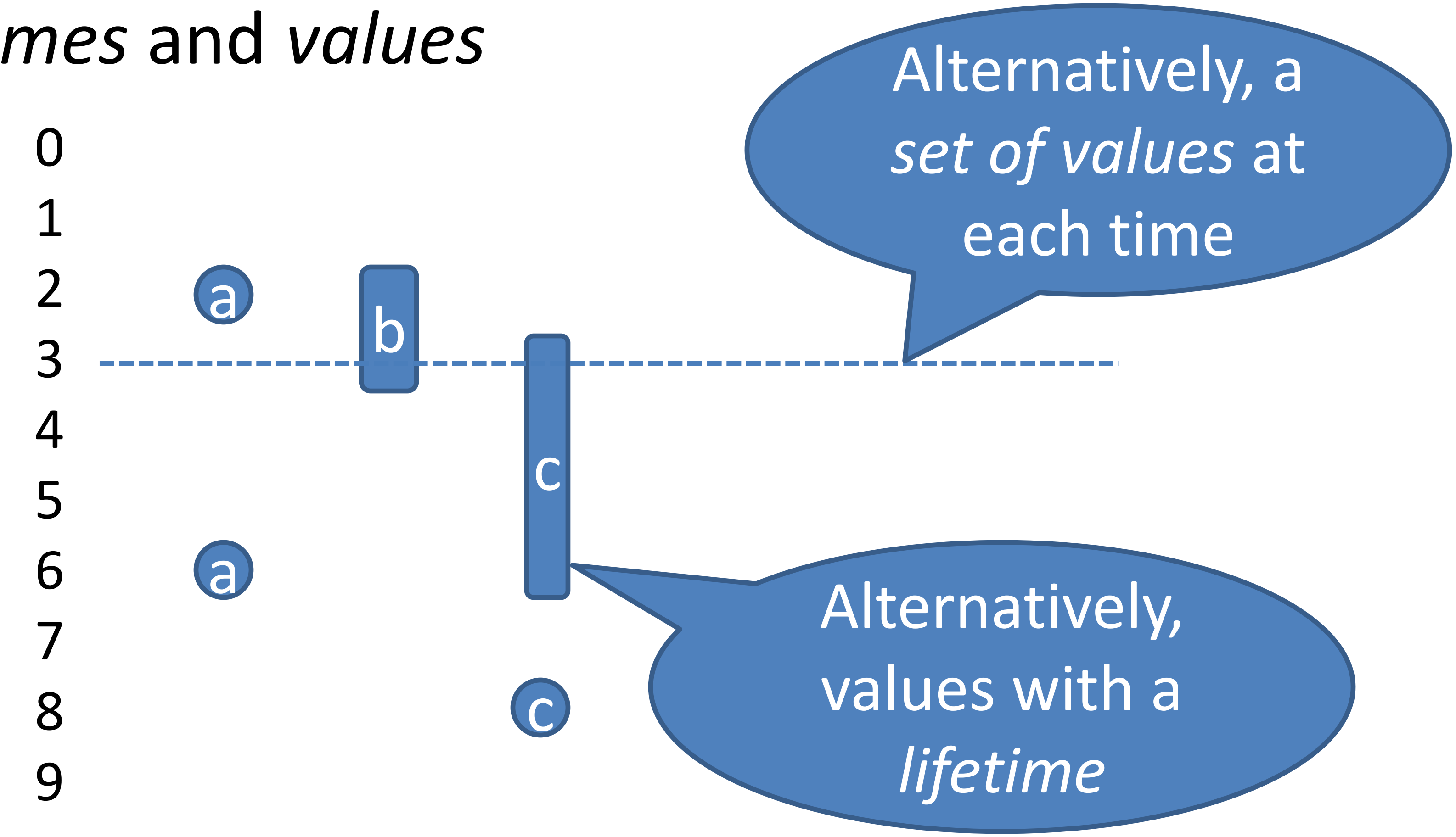
- Multiple correct behaviours
  - No ”expected results”

- Observed events not recorded atomically
  - Inaccurate times
  - Inaccurate order of events

- Complexity! Need a simple way to specify…

Temporal Relations

- A *temporal relation* is a relation between *times* and *values*



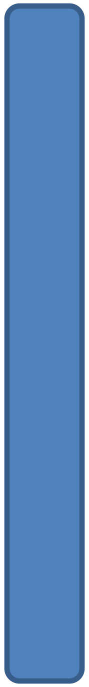
Alternatively, a *set of values* at each time

Alternatively, values with a *lifetime*

Example

Events as a temporal relation

10	{login,alice,laptop}
11	{login,bob,desktop}
15	{login,bob,phone}
26	{send,alice,bob,”Hi”}
31	{delivery,alice,bob,desktop,”Hi”}
33	{logout,bob,phone}



{logged_in,
bob,
phone}

States as a temporal relation

Logged-in Users

```
LoggedIn = stateful(fun logging_in/1,
                    fun logging_out/2,
                    Events)
```

- Start a state on a matching event

```
logging_in({login,Uid,ResourceId}) ->
  [{logged_in,Uid,ResourceId}].
```

- Transform a state on a matching event

```
logging_out({logged_in,Uid,Rid},Ev) ->
  case Ev of
    {logout,Uid,Rid} -> [];
    {unregister,Uid} -> []
  end.
```

Message Creations

Apply this function…

…to every pair of an event and logged-in user

- A ent to e user is lo ged in

```
MessageCreations =
   map(fun message_creation/1,
       product(Events,LoggedIn))
```

```
message_creation({{send,From,To,Msg},
                  {logged_in,To,Rid}}) ->
  {message,From,To,Rid,Msg}.
```

Messages in flight

```
Messages = stateful(fun start_message/1,
                    fun stop_message/2,
               union(MessageCreations,
                     Events))
```

```
start_message({message,From,To,R,Msg}) ->
  [{message,From,To,R,Msg}].
```

```
stop_message({message,From,To,R,Msg},Ev) ->
  case Ev of
    {delivery,From,To,R,Msg} -> [];
    {logout,To,R}            -> [];
    {unregister,To}          -> []
  end.
```

Message Delivery Deadline

- A relation containing messages overdue for delivery…

```
Overdue = all_past(100,Messages)
```

  – In flight for the last 100 ms

R

X

- In the property, check

```
is_empty(Overdue)
```

all_past(N,R)

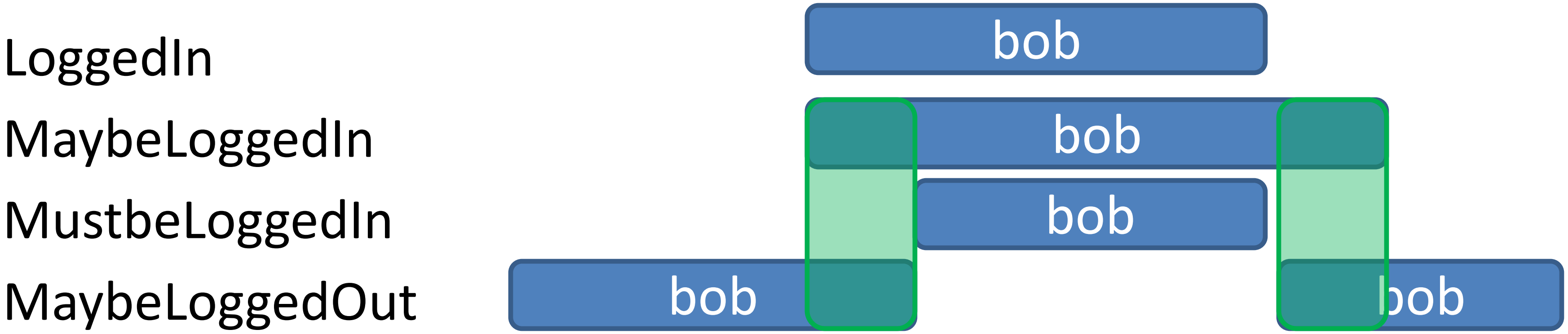
X

N

Timing Uncertainty

- If a user logs in on a second resource *just before* a message is sent, it need not be delivered…login may not be complete

```
MaybeLoggedIn   = any_past(15,LoggedIn),
MustbeLoggedIn  = all_past(15,LoggedIn),
MaybeLoggedOut  = complement(MustbeLoggedIn)
```



How well did it work?

- ~300 LOC replaced ad hoc version
- New spec was more modular and declarative
  - E.g. Messages *may* be delivered after a logout—for a short time
    - Old: needed 26 LOC at 4 separate locations
    - New: MaybeLoggedIn
  - E.g. Message delivery deadline
    - Old: appears in 5 places
    - New: OverdueMessages

We even found bugs!

- Send M to Bob & Bob logs in close together
  - M *should* be delivered to Bob
  - M only delivered on Bob's *next* login

- Send M to Bob & Bob logs out close together
  - M *should* be delivered to Bob now, or on next login
  - M may be lost altogether

Summary

- Race conditions *require* property-based testing
  - Serializability is an effective property to use
  - Temporal relations express asynchronous properties simply

- QuickCheck makes it easy to find concurrency bugs that have lurked in production code for years